Sveučilište Jurja Dobrile u Puli
Fakultet Informatike

GORAN VINKOVIĆ

RAZVOJ ATMOSFERIČNE 2D PUZZLE PLATFORMER IGRE U UNITY OKRUŽENJU

Diplomski rad

Pula, rujan, 2019.

Sveučilište Jurja Dobrile u Puli
Fakultet Informatike

GORAN VINKOVIĆ

RAZVOJ ATMOSFERIČNE 2D PUZZLE PLATFORMER IGRE U UNITY OKRUŽENJU

Diplomski rad

JMBAG: 0303038043, redoviti student

Studijski smjer: Informatika

Predmet: Dizajn i programiranje računalnih igara

Znanstveno područje: Društvene znanosti
Znanstveno polje: Informacijske i komunikacijske znanosti
Znanstvena grana: Informacijski sustavi i informatologija

Mentor: doc. dr. sc. Tihomir Orehovački

Pula, rujan, 2019.

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Goran Vinković, kandidat za magistra INFORMATIKE ovime izjavljujem da je ovaj Diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Diplomskog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

Goran Vinković

U Puli, 18.09., 2019 godine

IZJAVA

o korištenju autorskog djela

Ja, Goran Vinković dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom Razvoj atmosferične 2D puzzle platformer igre u Unity okruženju _____ koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, 18.09.2019 (datum)

Potpis

Vinković Goran

SADRŽAJ

UVOD

Tema ovoga rada je izrada 2D puzzle platformera u Unity razvojnome okruženju. Unity je među-platformsko razvojno okruženje za razvoj igara koje je razvijeno od strane Unity Technologies. Verzija programa koja se koristi za izradu ovoga projekta i rada jest Unity 2019.1.

Kroz ovaj rad prikazan je proces izrade igre Transcendence, od procesa animiranja lika do kodiranja različitih mehanika unutar igre. Programski jezik koji se koriti za izradu projekta jest C#.

Neke od igara koje inspiriraju dizajn igre Transcendence su; Celeste, Ori and the Blind Forest i Hollow Knight. Celeste je 2D stilizirana video igra od strane kanadskih programera Matta Thorsona i Noela Berryja, s umjetnošću brazilskog studija MiniBoss.

Ori and the Blind Forest je platformerska Metroidvania avantura razvijena od strane Moon Studios i objavljena od strane Microsoft Studios. Hollow Knight je 2D akcijski platformer razvijen i objavljen od strane studija Team Cherry.

Kako bi prešao razine igrač mora proći kroz zagonetke i izbjegavati zamke i smrtonosni okoliš, pri čemu koristi svoje sposobnosti kretanja na kojima je ujedno i glavni naglasak kao glavna mehanika unutar igre.

Prilikom igranja igre Transcendence igrač može kontrolirati više formi glavnoga lika. Druga forma osim normalne humanoidne uključuje formu mačke. Forma mačke je mala i omogućuje prolaz kroz uske prostore.
U igri Transcendence jedan od najbitnijih elemenata igre jest kretanje lika, pošto je unutar igre fokus na glatko i brzo kretanje, izbjegavanje zamki i skakanje po zidovima i platformama.
U radu se prikazuje proces izrade, odnosno kodiranja mehanika igre te implementacija istih. Igra se sastoji od 5 razina. Mehanike prikazane u radu su; kretanje, detektiranje kolizije, dijalozi, zamke unutar igre.
Dobra mehanika kretanja je sastavni dio igre te obuhvaća akcije kao što su; skakanje sa zida na zid, penjanje po zidovima, držanje za zid, bolja kontrola akcije skoka i naglih poleta na komandu igrača.

1. RAZVOJNO OKRUŽENJE

Igra Transcendence, odnosno projekt ovoga rada izrađen je u razvojnom okruženju Unity Engine. Unity je među-platformsko razvojno okruženje za razvoj igara koje je razvijeno od strane Unity Technologies, prvi put najavljeno i objavljeno u lipnju 2005. godine na Appleovoj Worldwide Developers konferenciji kao Mac OS X ekskluzivni alat (Axon, 2016).

Od 2018. alat je proširen na više od 25 platformi. Unity se može koristiti za stvaranje trodimenzionalnih igara, dvodimenzionalnih igara, igara virtualne stvarnosti i igara proširene stvarnosti, kao i simulacija i drugih iskustava. Unity je usvojen od strana industrije izvan video igara, kao što su filmska industrija, automobilska industrija, arhitektura, inženjering i izgradnja. Nekoliko glavnih verzija Unityja je objavljeno od njegovog pokretanja. Najnovija stabilna verzija, 2019.1.10, objavljena je u srpnju 2019. godine (Axon, 2016).

Unity korisnicima daje mogućnost stvaranja igara i iskustava u 2D i 3D tehnologiji, a alat nudi primarni API za skriptiranje u C#, i za alat Unity i u obliku dodataka i samih igara.. Prije C# kao primarnog programskog jezika koji se koristio za alat, Unity je prethodno podržavao Boo, koji je uklonjen izdavanjem Unity 5, i verziju JavaScripta pod nazivom UnityScript, koja je zastarjela u kolovozu 2017., nakon objavljivanja Unity 2017.1, u korist C# (Unity Technologies, 2019).

Unutar 2D igara, Unity dopušta uvoz spriteova i ima napredni 2D svjetski prikazivač (eng. renderer). Za 3D igre, Unity omogućuje specifikaciju kompresije teksture, mipmapa i postavki rezolucije za svaku platformu koju podržava igraći alat, i pruža podršku za mapiranje bump-ova, mapiranje refleksije, mapiranje paralaksa, okluziju okolnog prostora ekrana (SSAO), dinamičke sjene pomoću mapa sjena, efekata prikaza u teksturi (eng. render-to-texture) i efekata naknadne obrade (eng. full-screen post-processing) (Unity Technologies, 2019).

Od 2018. godine, Unity je korišten za stvaranje otprilike polovice novih mobilnih igara na tržištu i 60 posto igara proširene stvarnosti i sadržaja virtualne stvarnosti.

2. SLIČNE IGRE

Transcendence je kao 2D platformer igra uvelike inspirirana od strane igara kao što su Celeste, Ori and the Blind Forest, Hollow Knight. U nastavku slijedi opis navedenih igara i pregleda igrivosti istih.

2.1 Celeste

Celeste je 2D stilizirana video igra od strane kanadskih programera Matta Thorsona i Noela Berryja, s umjetnošću brazilskog studija MiniBoss. Igra je originalno kreirana kao prototip u četiri dana tijekom Game Jam-a, te je kasnije proširena u potpuno izdanje. Celeste je izdana u Siječnju 2018. godine na platformama Microsoft Windows, Nintendo Switch, PlayStation 4, Xbox One, macOS i Linux (Wikipedia, 2019).

Celeste je platformska igra u kojoj igrači kontroliraju djevojku po imenu Madeline kojoj je cilj proći kroz planinu dok izbjegava razne smrtonosne prepreke. Uz skakanje i penjanje po zidovima na limitirano vrijeme, Madeline ima sposobnost izvođenja lansiranja / polijetanja u zrak (eng. dash) u osam kardinalnih i interkardinalnih smjerova (Wikipedia, 2019).

Taj se potez može izvesti samo jednom i mora se ponovno napuniti bilo slijetanjem na tlo, udaranjem određenih predmeta kao što je obnavljajući kristal, ili premještanjem na novi ekran (iako je igraču kasnije dodijeljeno dodatno lansiranje u igri). (Wikipedia, 2019).

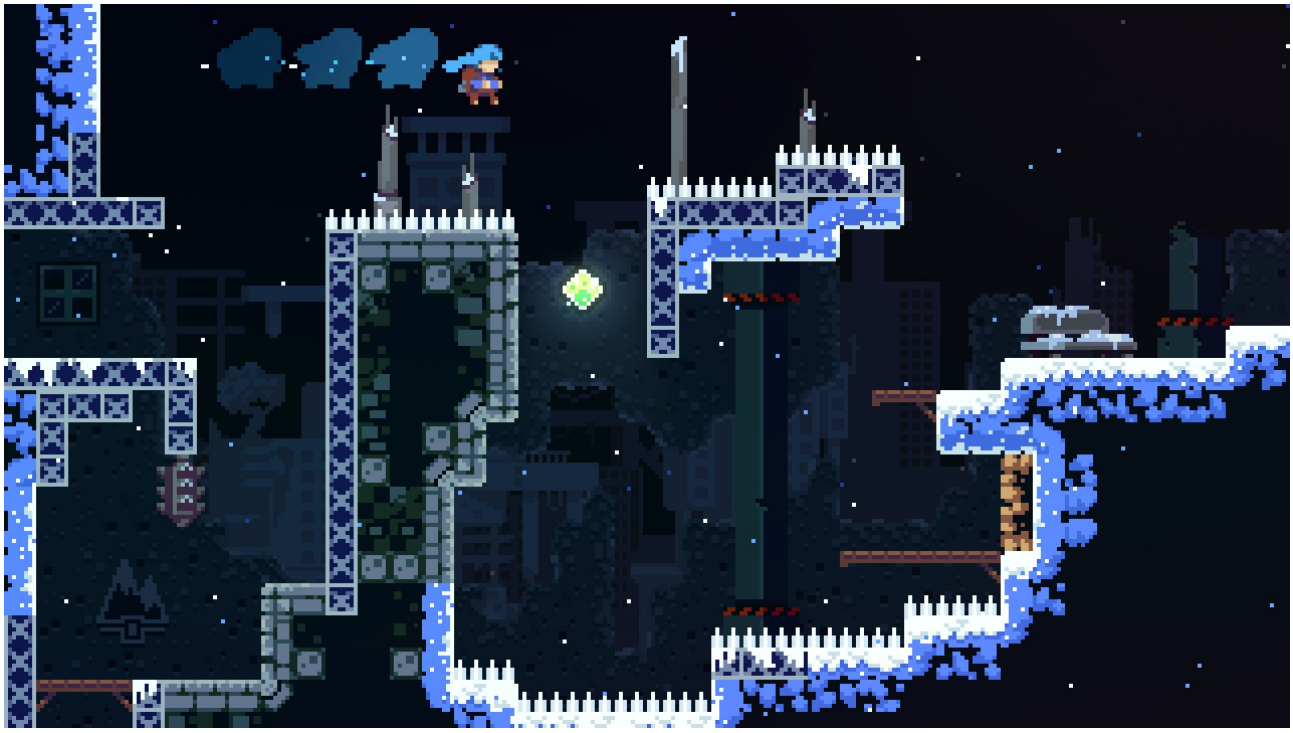
Tijekom svake razine, igrač će susresti dodatne mehanike, kao što su opruge koje lansiraju igrača ili perje koje omogućuje kratak let, i smrtonosne predmete poput šiljaka koji ubijaju Madeline (vraćajući je na početak razine) (Wikipedia, 2019).

Igrači također mogu pristupiti modu pomoći, gdje mogu promijeniti neke atribute o fizici igre. Neki od njih uključuju: beskonačna zračna lansiranja, nepobjedivost ili usporavanje brzine igre (Wikipedia, 2019).

Skrivene tijekom svake razine su neobavezne jagode, dobivene kroz teške platformske izazove ili slagalice, koje malo utječu na završetak igre, ovisno o tome koliko ih je prikupljeno. Osim toga, postoje kazete koje otključavaju teže "B-Strane" (eng. B-Side) varijacije određenih razina, i kristalna srca koja se koriste za pristup sadržaju nakon igre (Matulef, 2016).

Prelaženje svih "B-strana" zatim otključava "C-Strane", koje se sastoje od vrlo teških, ali kratkih varijacija na razinama. Nakon prelaska svih "C-strana", igrač može pristupiti izborniku Varijante. Izbornik Varijante omogućuje igračima da promijene fizičku igru na način sličan načinu moda pomoći (Matulef, 2016).

Neke od tih "varijantnih" postavki uključuju: ubrzavanje igre, lansiranje od 360 stupnjeva i nisko trenje za sve ravne površine. Ove postavke služe kako bi igra bila izazovnija ili zabavnija. Originalni prototip Celeste Classic Pico-8 također se može naći kao skrivena mini igra (Matulef, 2016).



Slika 1: Snimka ekrana iz igre Celeste, izvor; https://steamcdn-a.akamaihd.net/steam/apps/504230/ss_4b0f0222341b64a37114033aca9994551f27c161.jpg?t=1567740791, pristupljeno 18.09.2019

2.2 Ori and the Blind Forest

Ori and the Blind Forest je platformerska Metroidvania avantura razvijena od strane Moon Studios i objavljena od strane Microsoft Studios. Igra je izdana za Microsoft Windows i Xbox One 11. ožujka 2015 (Wikipedia, 2019).

U igri, igrači preuzimaju kontrolu nad Ori, bijelim duhom čuvarom, i Seinom, "svjetlom i očima" drevnog šumskog stabla. Za napredak u igri, igrači imaju zadatak da se kreću između platformi i rješavaju zagonetke. Igra ima sustav pod nazivom "veza duše", koji omogućuje igračima spremanje stanja igre po volji, i nadogradnju sustava koji daje igračima sposobnost da ojačaju Orine vještine (Wikipedia, 2019).

Igru je razvio Moon Studios. Igru je Microsoft Studios kupio godinu dana nakon početka razvoja igre. Priča u igri bila je inspirirana Kraljem Lavova (eng. Lion King) i Željeznim Divom (eng. The Iron Giant), dok su neki od elemenata igranja bili inspirirani Raymanovim i Metroidovim franšizama (Wikipedia, 2019).

Nakon objavljivanja, igra je dobila pohvale kritičara, s igračima koji su hvalili igru, umjetnički stil, priču, akcijske sekvence, glazbeni i okolišni dizajn. Suosnivač tvrtke Moon Studios Gennadiy Korol izjavio je kako je igra bila profitabilna za tvrtku u roku od nekoliko tjedana nakon početnog izdavanja (Wikipedia, 2019).

Ori može skakati, penjati se i koristiti druge sposobnosti za navigaciju. Sein može pucati vatrene projektile u borbi protiv neprijatelja ili razbijati prepreke. Ori je potreban za interakciju s okolinom dok skaču s platformi i rješavaju zagonetke (Xbox News, 2019).

Ori je suočen s neprijateljima dok se probija kroz svijet kako bi obnovio šumu. Igrač pomaže Ori da sakupi krhotine zdravlja, krhotine energije, nove sposobnosti i nadogradnje. Radnja igre se odvija na klasičan način igara podžanra Metroidvania, sa stjecanjem novih sposobnosti koje omogućuju igraču pristup prethodno nepristupačnim područjima (IGN, 2019).

Osim uštede bodova raspršenih u igri, igrači mogu stvoriti "duševne veze" u bilo koje vrijeme kada odaberu da služe kao kontrolne točke. Međutim, veze duše mogu se stvoriti samo pomoću energetskih stanica prikupljenih tijekom igranja; potrebna energija nije u izobilju i prisiljava igrače da ih stvaraju samo kad je to potrebno (IGN, 2019).

Igrač dobiva sposobnost trošenja bodova za kupnju različitih pogodnosti i nadogradnji, kao što je povećanje štete od Seinovog plamena. Ove nadogradnje mogu se kupiti bilo gdje je stvorena veza duše i ako igrač ima dovoljno bodova sposobnosti da kupi vještinu koju želi. Točka sposobnosti stječe se kada Ori prikupi dovoljno iskustva ubijanjem neprijatelja i uništavanjem raznih biljaka. Svaka vještina mora se kupiti u nizu od jednog od tri stabla sposobnosti kako bi se omogućilo da sljedeća, skuplja vještina bude dostupna (IGN, 2019).

Slika 2: Snimak ekrana iz igre Ori and the Blind Forest, izvor https://steamcdn-a.akamaihd.net/steam/apps/387290/ss_c1a7eb159190ffc77af529ea99ae81365c354312.jpg?t=1557766574 , pristupljeno 18.09.2019

2.3 Hollow Knight

Hollow Knight je 2D akcijski platformer razvijen i objavljen od strane studija Team Cherry. Igra je izdana za Microsoft Windows, MacOS i Linux u 2017., a za Nintendo Switch,

PlayStation 4, Xbox One 2018. godine. Razvoj je djelomično financiran kroz Kickstarter kampanju grupnog financiranja, što je dovelo do povećanja od 57.000 USD do kraja 2014 (Kickstarter, 2019).

Igra priča priču o vitezu u potrazi za otkrivanjem tajni davno napuštenog kraljevstva kukaca Hallownest, čije zastrašujuće dubine privlače avanturiste i hrabre s obećanjima blaga i odgovorima na drevne tajne (Kickstarter, 2019).

Igrač kontrolira viteza sličnoga kukcu. Tihi i bezimeni vitez istražuje podzemni svijet. Vitez posjeduje mač zvani Nail, koji je u obliku stošca, koji se koristi u borbi i interakciji s okolišem (Kickstarter, 2019).

U većini područja igre igrač susreće neprijateljske bube i druge vrste bića. Bliska borba uključuje korištenje mača da udari neprijatelje s kratke udaljenosti. Igrač također može naučiti magiju, dopuštajući napade na velike udaljenosti. Poraženi neprijatelji bacaju valutu pod nazivom Geo (Kickstarter, 2019).

Vitez počinje s ograničenim brojem maski, koje predstavljaju bodove života lika. Krhotine maski mogu se prikupiti tijekom igre kako bi se povećalo zdravlje igrača. Kada vitez primi štetu od neprijatelja ili iz okoline, oduzima se maska (Wikipedia, 2019).

Udaranjem neprijatelja, vitez stječe dušu, koja se nalazi u spremniku duša. Ako su sve maske izgubljene, vitez umire i na tom se mjestu pojavi Sjena. Igrač također gubi sve Geo novčiće i može zadržati smanjenu količinu duše (Wikipedia, 2019).

Igrač mora poraziti Sjenu kako bi povratio izgubljenu valutu i kako bi opet mogao nositi normalnu količinu duše. Igra se nastavlja s posljednje posjećene klupe - raštrkane su po cijelom svijetu igre i djeluju kao točke spašavanja. U početku igrač može samo koristiti duše za "Focus" (regeneriranje maski), ali kako igra napreduje, igrač otključava nekoliko ofenzivnih čarolija, koje konzumiraju duše kao resurs (Wikipedia, 2019).

Mnoga područja imaju više izazovnih neprijatelja (šefova) koje igrač može poraziti kako bi napredovao dalje. Pobjeđivanje nekih šefova daje igračima nove sposobnosti. Kasnije u igri igrač dobiva legendarnu oštricu koja može "probiti veo između snova i buđenja". To

omogućuje igraču da se suoči s izazovnijim verzijama nekih šefova, te da razbije ono što zatvara put do konačnog šefa (Wikipedia, 2019).

Tijekom igre, igrač susreće likove koji nisu igrači, odnosno NPC likove (eng. non-playable characters) s kojima može komunicirati. Ti likovi pružaju informacije o priči unutar igre, nude pomoć i prodaju predmete ili usluge. Igrač može nadograditi vitezov mač kako bi radio više štete ili pronašao spremnike duše za nošenje više duša. Tijekom igre igrač stječe predmete koji pružaju nove pokretne sposobnosti. Oni uključuju dodatni skok u zrak (predmet naziva Monarch Wings), prianjanje uz zidove i skakanje s njih (predmet naziva Mantis Claw), i brzo pomicanje (Wikipedia, 2019).

Igrač može naučiti i druge borbene sposobnosti, poznate kao umjetnost mača, i spomenute magije. Da bi se vitez dodatno prilagodio, igrač može opremiti razne privjeske, koji se mogu naći ili kupiti od NPC-a (Wikipedia, 2019)

Neki od njihovih učinaka uključuju: poboljšane borbene sposobnosti ili vještine, više maski ili njihovu obnovu, bolje vještine kretanja, lakše prikupljanje valute ili duše, i transformaciju. Opremanje privjesaka zauzima određeni broj ograničenih mjesta, nazvanih urezima. Moguće je nošenje privjeska koji zahtijeva više od dostupnog broja ureza, ali rezultira time da je lik "overcharmed", uzrokujući da vitez primi dvostruku štetu od svih izvora (Wikipedia, 2019)

Hallownest se sastoji od nekoliko velikih, međusobno povezanih područja s jedinstvenim temama. Iako Hollow Knight ne veže igrača na jednu stazu kroz igru, niti zahtijevaju od njih da istražuju cijeli svijet, postoje prepreke koje ograničavaju pristup igraču. Igrač će možda morati napredovati u priči o igri, ili će steći određenu sposobnost kretanja, vještinu ili predmet za daljnji napredak (Wikipedia, 2019).

Za brzo putovanje kroz svijet igre, igrač može koristiti "Stag" postaje, terminale mreže tunela; igrač može putovati samo do prethodno posjećenih i otključanih stanica. Ostale metode brzog putovanja, tramvaji i "Dreamgate" susreću se kasnije u igri (Whitaker, 2017).

Kako igrač ulazi u novo područje, on nema pristup karti okoline. Moraju pronaći Cornifera, kartografa, kako bi kupili grubu kartu. Kako igrač istražuje područje, karta postaje

preciznija i potpuna, iako se ažurira samo kada sjedi na klupi. Igrač će morati kupiti određene predmete kako bi dovršio karte, vidio točke interesa i postavio oznake. Položaj viteza na karti može se vidjeti samo ako igrač nosi određeni privjesak (Whitaker, 2017).

Slika 3: Snimak ekrana iz igre Hollow Knight, izvor; https://steamcdn-a.akamaihd.net/steam/apps/367520/ss_62e10cf506d461e11e050457b08aa0e2a1c078d0.jpg?t=1568794924 , pristupljeno 18.09.2019

3. RADNJA IGRE

Naziv igre u sklopu ovoga rada jest Transcendence. Riječ transcendence jest engleski izraz za postojanje, odnosno iskustvo van normalno fizičkog svijeta. Točnije, spiritualno iskustvo.

Unutar igre, igrač je postavljen u ulogu avanturista koji se nađe u zagrobnome svijetu. Doduše, tu je sasvim slučajno, najvjerojatnije greškom nekog šeprtljavog boga. Na početku igre, igrač upoznaje NPC-a koji ga upoznaje s trenutnom situacijom.

Pošto je igrač ovdje greškom i nije zapravo napustio stvarni svijet, odnosno umro, i dalje je živ te ne može na put kojim idu duše preminulih. Igrač je vrlo interesantan slučaj, iako je živ ali se nalazi u zagrobnome svijetu, dobio je pristup određenim moćima. Dobio je mogućnost nadnaravnog kretanja i mijenjanja formi. Druga forma osim normalne humanoidne uključuje formu mačke. Forma mačke je mala i omogućuje prolaz kroz uske prostore.

Kako bi se igrač vratio u svijet živih mora doći do bunara života zvanog Hvergelmir i piti iz njega. Vjeruje se kako je taj bunar izvor svog života i ima mogućnost vraćanja igrača natrag u život.

Između igrača i bunara života stoji nekoliko negostoljubivih svjetova:

- Niflheimr – prva razina nakon početne razine. Taj svijet je inspiriran svijetom zvanim Niflheim iz nordijske mitologije. Vrlo hladno mjesto, puno leda i tame.
- Musphlheimer – drugia razina, svijet inspiriran svijetom zvanim Misphelheim iz nordijske mitologije. Svijet je praktički živi vulkan. Puno vatre, lave i kamenja.
- Vanerheimr – treća razina, svijet inspiriran svijetom zvanim Vanaheim iz nordijske mitologije. Tematika svijeta jest tamna, zla šuma.
- Midgrheimr – četvrta razina, svijet inspiriran verzijom zemaljskog svijeta Midgard iz nordijske mitologije. Normalna, prašuma u kojoj se ujedno nalazi i bunar života do kojega igrač mora doći.

Kako bi prešao razine igrač mora proći kroz zagonetke i izbjegavati zamke i smrtonosni okoliš, pri čemu koristi svoje sposobnosti kretanja na kojima je ujedno i glavni naglasak kao glavna mehanika unutar igre.

4. LIKOVI

U ovome poglavlju slijedi pregled likova koje igrač može igrati, te prikaz procesa kreiranja likova unutar Unity razvojnog okruženja, animiranja i snimanja njihovih animacija, kreiranje animatora koji kontrolira animacije i kreiranje kontrolera za likove.

Prilikom igranja igre Transcendence igrač može kontrolirati više formi glavnoga lika.

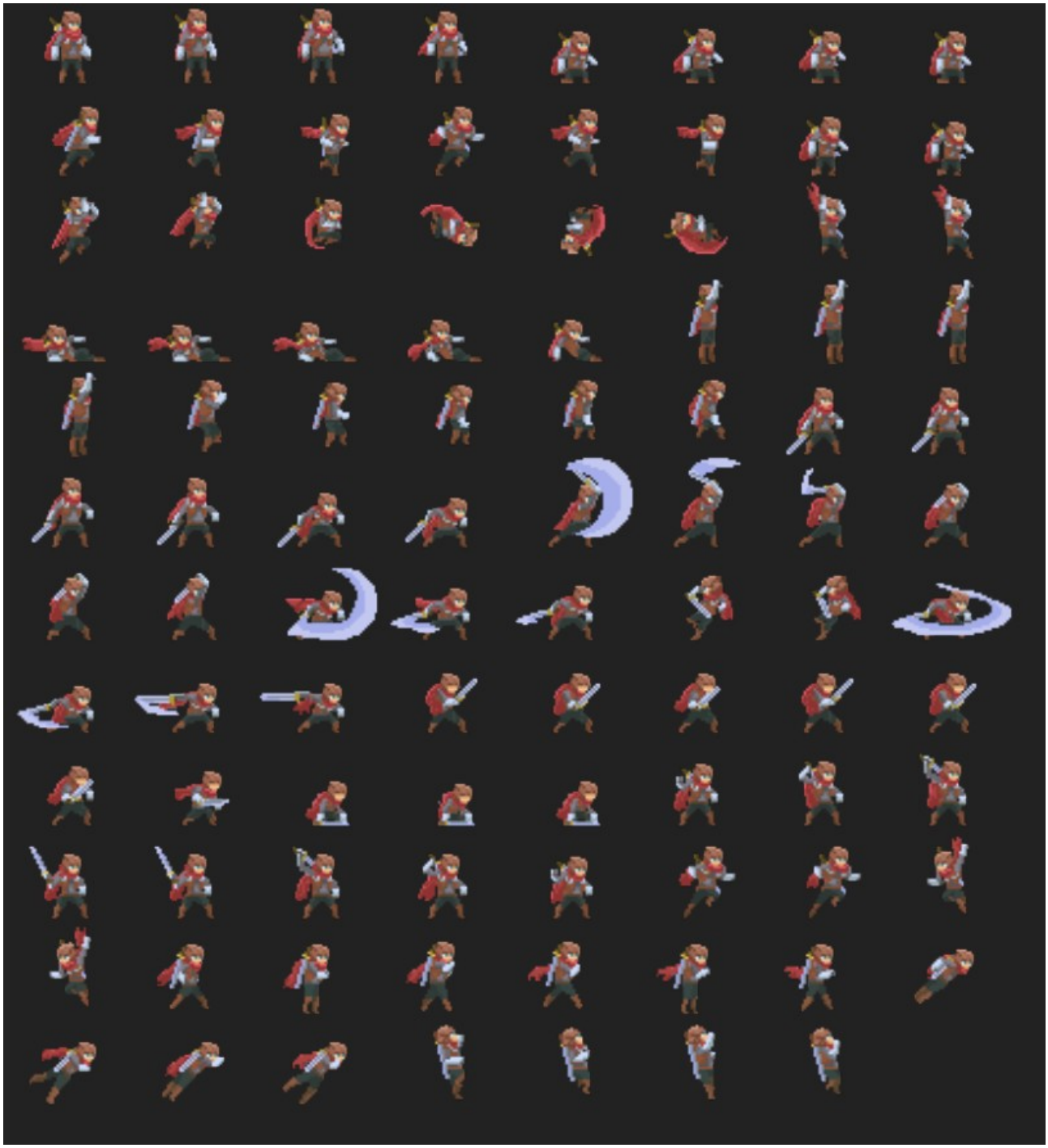
Pošto je Transcendence 2D igra, likovi nisu klasični 3D modeli, već slika (eng. sprite sheet) koja u sebi sadrži više manjih sličica (eng. sprite), što je vidljivo na slici 4 glavnoga lika.

Kako bi se dobile pojedine PNG slike unutar Unity razvojnog okruženja koristi se alat Sprite Editor. Sprite Editor je alat unutar Unity okruženja koji pruža mogućnost izvlačenja manjih slika iz kompozitne slike.

Prije nego li se slika može koristiti u alatu Sprite Editor potrebno je podesiti postavke importiranja (eng. import settings). Najvažnije jest namjestiti tip teksture na Sprite (2D i UI). Također, ako slika sadrži više elemenata potrebno je namjestiti mod slike (eng. sprite mode) na višestruko (eng. multiple).

Kompozitna slika (eng. sprite sheet) ponajviše koristi za smanjivanje memorije potrebne za sve pojedine slike. Na primjer, kod slike veličine 140 x 140 pixela, za svaki piksel potrebna je određena količina memorije za pohranjivanje njegove boje.

Točna veličina ovisi o dubini boje, koja je standardno 32-bitna i prema tome zauzima 4 bajta prostora. Prema tome, totalna memorija potrebna za pohranjivanje spomenute slike jest 140 x 140 x 4 što iznosi 76kB.



Slika 4: Kompozitna slika glavnoga lika, izvor: projekt

Ovisno o grafičkom hardveru, slike bi mogle zahtijevati određene veličine, npr., kvadratne veličine, odnosno, kvadratne slike (eng. sprites). Zbog toga slika mora biti popunjena s dodatnim neiskorištenim pikselima, kako bi odgovarala zahtjevima hardvera.

Za ispunjenje tih zahtjeva spomenutu sliku je potrebno proširiti na 256 x 256 piksela. To povećava korištenje memorije na 256kB, što je tri puta veće od originalne veličine slike.

Veličina nije velika za pojedinu sliku, ali kod kreiranja igre s puno objekata, likova i animacija vrlo lako se može koristiti više stotina slika. Kompozitna slika služi za pakiranje drugih slika u taj neiskorišteni prostor.

Alat Sprite Editor ima nekoliko načina za rezanje slike; automatski, po veličini ćelija u mreži (eng. grid by cell size), po broju ćelija u mreži (eng. grid by cell count) (Unity Documentation, 2019).

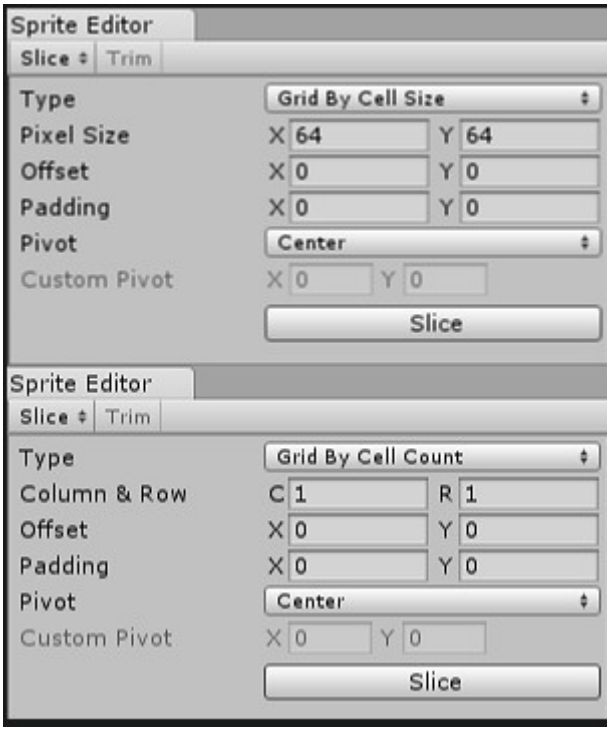
- Automatski – Editor automatski određuje granice elemenata slike po transparentnosti između njih. Može se postaviti standardna pivot pozicija za svaku identificiranu sliku. Ova metoda omogućava različito djelovanje s postojećim odabirima.

    - Brisanje postojećeg – ova opcija zamjenjuje što god je odabrano.
    - Pametno – nastoji stvoriti nove kvadrate dok pritom zadržava ili prilagođava postojeće.
    - Sigurno – stvara nove kvadrate bez izmjenjivanja postojećih.

- Po veličini ćelija u mreži – korisna opcija ako su elementi slike poredani u regularnome uzorku prilikom kreacije.
- Po broju ćelija u mreži – korisna opcija ako su elementi slike poredani u regularnome uzorku prilikom kreacije.

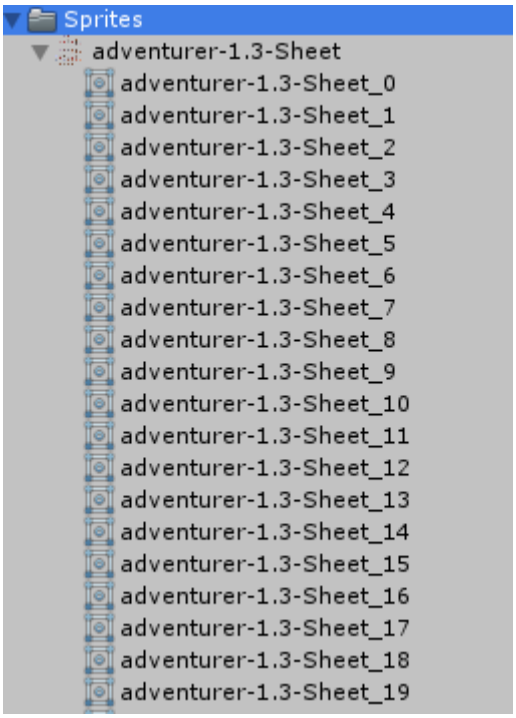
Na slici 5 su prikazane opcije Sprite Editor alata kod opcija veličina ćelija u mreži i broj ćelija u mreži.

Veličina piksela (eng. pixel size) je vrijednost koja određuje visinu i širinu elemenata slike u pikselima za opciju veličina ćelija u mreži. Stupac i red (eng. column and row) je vrijednost koja određuje broj stupaca i redaka za rezanje (Unity Documentation, 2019).



Slika 5: Prikaz opcija Sprite Editora kod opcija grid by cell size i grid by cell count, Izvor: Unity Editor

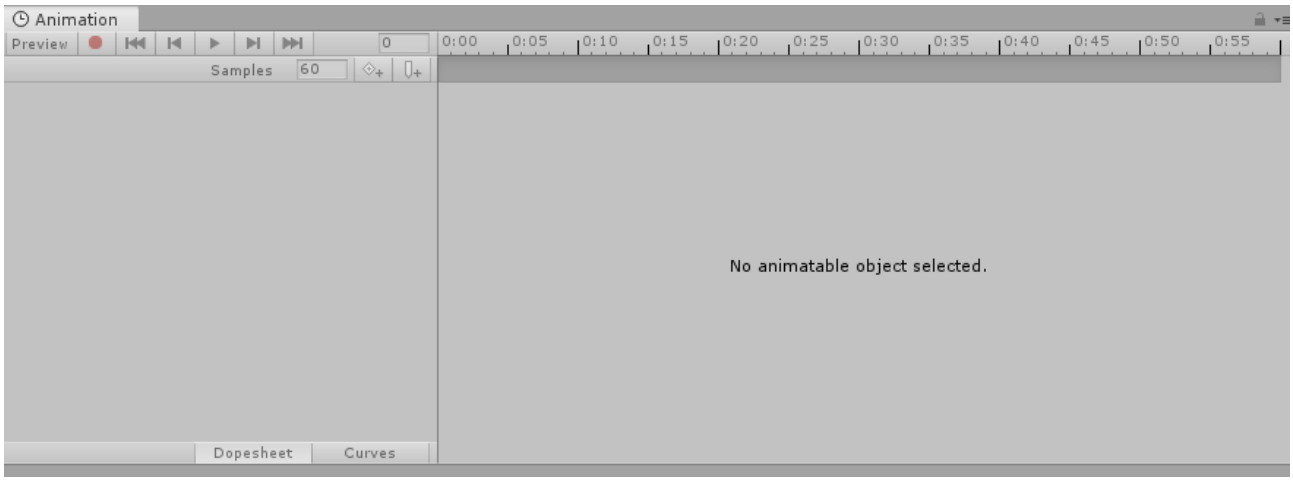
Za potrebe igre Transcendence korištena je opcija za automatsko rezanje. Kada se potvrdi odabir opcije Unity reže sliku, te izdvaja elemente unutar slike kao pojedine slike PNG formata unutar hijerarhije što je vidljivo na slici 6.



Slika 6: Prikaz hijerarhije slike nakon rezanja Sprite Editor alatom, izvor: Unity Editor

4.1 Animiranje lika

Za animiranje lika prvo je potrebno kreirati snimke (eng. clip) animacije. Za snimanje animacija koristi se Animation ugrađeni alat unutar Unity razvojnog okruženja.



Slika 7: Prikaz alata za snimanje animacija, Izvor: Unity Editor

Kod 2D igara proces animiranja je dosta jednostavniji nego kod 3D modela i igara, pošto se radi s 2D slikama. Za izradu, recimo, animacije hodanja potrebno je odabrati one slike lika koje sadrže takvu informaciju i jednostavno ih povući u Animation alat vidljiv na slici 7 (Unity Documentation, 2019).

Unity automatski prepoznaje da se radi o 2D slikama, te generira animaciju koja je zapravo sekvenca, odnosno, niz odabranih slika koje tvore animaciju. Svaki lik ili objekt kojega se želi animirati mora u svojim komponentama sadržavati komponentu Animator kontroler.

Animator kontroler je komponenta koja dozvoljava uređivanje i održavanje snimaka animacije i asociranih tranzicija animacija za likove ili objekte. U većini slučajeva normalno je imati više animacija i raditi prijelaze između istih kada se provedu određeni uvjeti unutar igre. Na primjer, može doći do promjene iz animacije šetanja (eng. walk) u animaciju skakanja (eng. jump) na pritisak Spacebar tipke na tipkovnici (Unity Documentation, 2019).

Animator kontroler sadrži reference na snimke animacija koje se koriste unutar njega, i upravlja raznim animacijama i tranzicijama između njih koristeći se konačnim automatima stanja (eng. state machine), koji služe kao dijagram stanja za animacije i tranzicije (Unity Documentation, 2019).

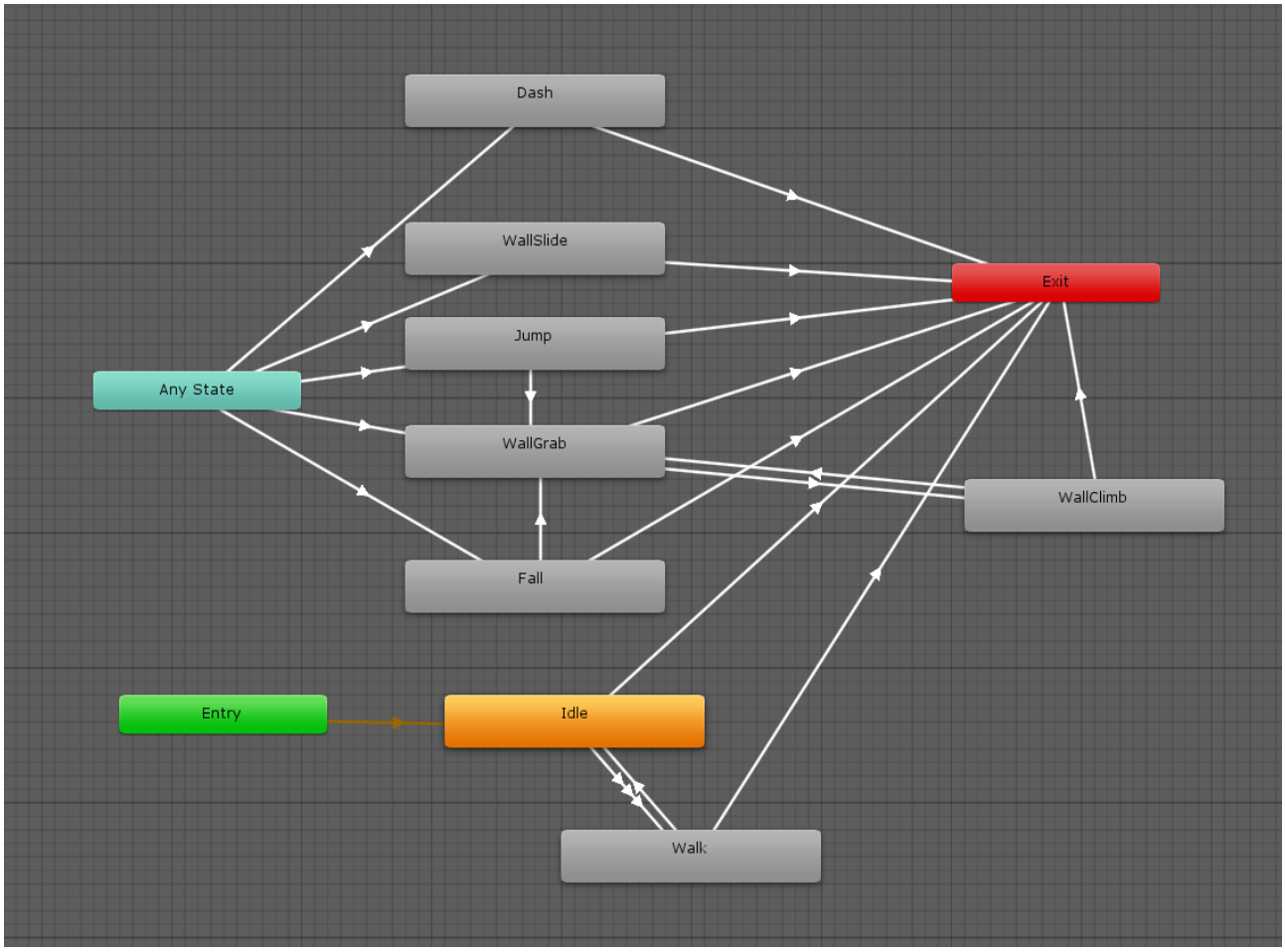
Akcije tijekom igre kao što su hodanje, trčanje, skakanje se nazivaju stanja, u smislu da je neki lik ili objekt u stanju gdje hoda ili radi nešto drugo. Uobičajeno, lik ima nekakve restrikcije za prijelaz na sljedeće stanje na koje može ići, umjesto da može trenutno mijenjati stanja iz bilo kojeg u bilo koje drugo stanje (Unity Documentation, 2019).

Na primjer, skok iz trka se može izvesti samo i samo kada lik već trči i kada ne stoji na mjestu, prema tome ne bi se nikada trebao dozvoliti prijelaz iz stanja mirovanja u stanje skoka iz trka. Opcije za sljedeće stanje u koje lik može prijeći iz trenutnog stanja se nazivaju tranzicije stanja. Zajedno, set stanja, set tranzicija i varijabli trenutnih stanja čine automat konačnih stanja (Unity Documentation, 2019).

Stanja i tranzicije automata konačnih stanja se reprezentiraju dijagramom, gdje čvorovi reprezentiraju stanja, a poveznice (strelice između čvorova) reprezentiraju tranzicije kao što je prikazano na slici 8. (Unity Documentation, 2019).

Važnost automata konačnih stanja je u tome što za animaciju mogu biti dizajnirani i nadograđeni vrlo lako. Svako stanje ima nekakav pokret, odnosno animaciju asociranu s njime koja će se odigrati kada automat stanja dođe u to stanje. To omogućava animatorima i dizajnerima definiranje sekvence akcija nekoga lika na vrlo jednostavan način (Unity Documentation, 2019).

Na slici 8 se nalazi prikaz animator kontrolera za igru Transcendence.



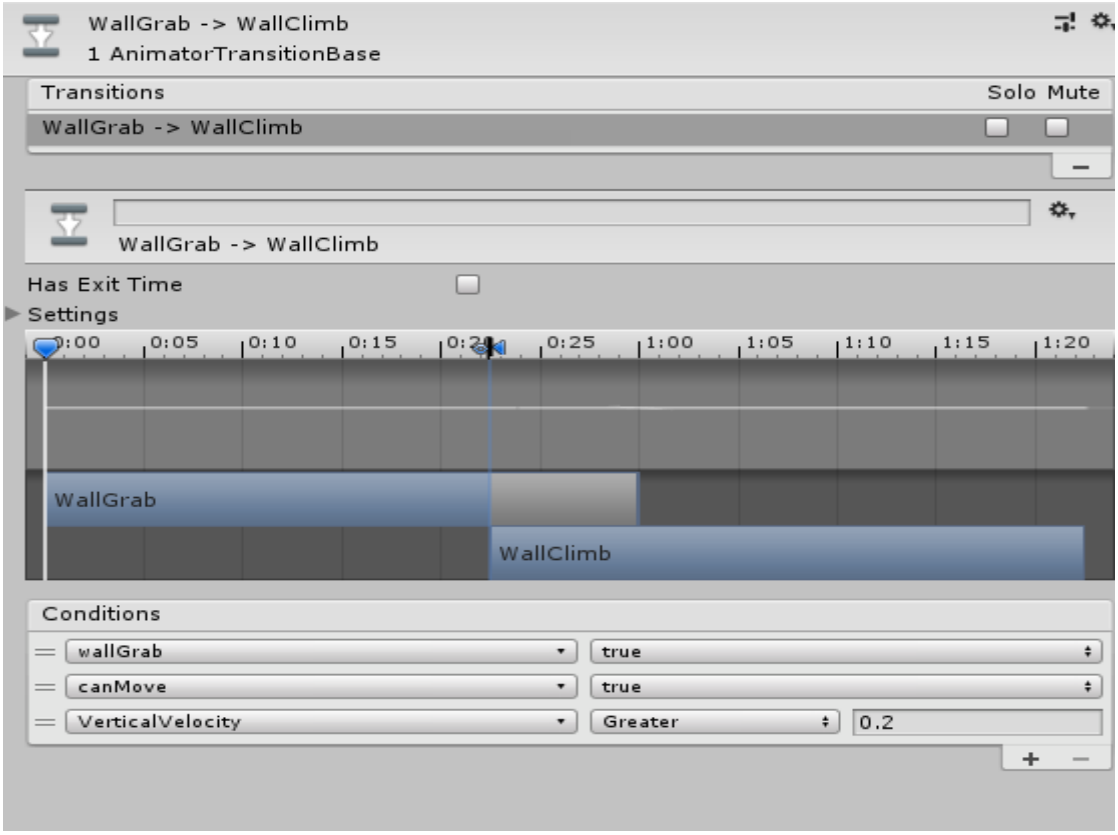
Slika 8: Prikaz animator kontrolera za glavnoga lika igre Transcendence, izvor: Unity Editor

Tranzicije između animacija omogućavaju automatu konačnih stanja da mijenja stanje animacije iz jednog u drugo ili da miješa animacije. Tranzicije ne definiraju samo koliko će miješanje između stanja animacija trajati i pod kojim uvjetima će se aktivirati. Na slici 9 je prikazan primjer tranzicije između stanja wallClimb i wallGrab (Unity Documentation, 2019).

U jednome trenutku može biti aktivna samo jedna tranzicija. Doduše, trenutno aktivna tranzicija može biti prekinuta od strane druge tranzicije ako je namješteno da se može prekinuti (Unity Documentation, 2019).

Kako bi se animacije izvodile glatko i prekidale pravovremeno ovisno o komandama koje igrač stišće na tipkovnici potrebno je pravilno namjestiti varijablu Has Exit Time. Izlazno

vrijeme (eng. exit time), je posebna tranzicija koja se ne obazire na parametre. Umjesto toga, obazire se na normalizirano vrijeme stanja. Ako se opcija uključi tranzicija će se dogoditi u specifično vrijeme određeno u opciji izlazno vrijeme (Unity Documentation, 2019).



Slika 9: Prikaz tranzicije između stanja wallGrab i wallClimb, izvor: Unity Editor

Odnosno, ako je opcija uključena animacija će se uvijek izvesti do kraja neovisno o tome ako igrač unese komandu koja pokreće drugo stanje. U platformerima se ta opcija najčešće isključuje kako bi se animacije mogle prekinuti i kako bi se odmah moglo prijeći na iduće stanje. To omogućuje glatko i responzivno kretanje što je iznimno bitno u 2D platformerima.

Sljedeća skripta prikazuje namještanje parametara potrebnih za okidanje animacija kroz kod. Parametri su vidljivi na slici 15.

```
public class AnimationScript : MonoBehaviour
{
    private Animator anim;
    private Movement move;
    private Collision coll;

    [HideInInspector]
    public SpriteRenderer sr;

    void Start()
    {
        anim = GetComponent<Animator>();
        coll = GetComponentInParent<Collision>();
        move = GetComponentInParent<Movement>();
        sr = GetComponent<SpriteRenderer>();
    }
```

Slika 10: Skripta AnimationScipt.cs , varijable i Start funkcija

Unutar Start() funkcije dohvaćaju se komponente potrebne skripti. Komponente su animator koji je potreban za mijenjanje parametara i okidanje animacija, Collision i Movement skripte koje su odgovorne za detektiranje kolizije i kretanje lika, te SpriteRenderer komponenta za animiranje slika lika.

```
void Update()
{
    anim.SetBool("onGround", coll.onGround);
    anim.SetBool("onWall", coll.onWall);
    anim.SetBool("onRightWall", coll.onRightWall);
    anim.SetBool("wallGrab", move.wallGrab);
    anim.SetBool("wallSlide", move.wallSlide);
    anim.SetBool("canMove", move.canMove);
    anim.SetBool("isDashing", move.isDashing);
}
```

Slika 11: Skripta AnimationScript.cs, Update funkcija

Unutar Update() funkcije postavljaju se Bool parametri ovisno o parametrima iz Collision i Movement skripti.

```
public void SetHorizontalMovement(float x, float y, float yVel)
{
    anim.SetFloat("HorizontalAxis", x);
    anim.SetFloat("VerticalAxis", y);
    anim.SetFloat("VerticalVelocity", yVel);
}
```

Slika 12: Skripta AnimationScript.cs, SetHorizontalMovement funkcija

SetHorizontalMovement funkcija prima float parametre i predaje iste animatoru koji na temelju primljenih vrijednosti postavlja i okida animacije.

```
public void SetTrigger(string trigger)
{
    anim.SetTrigger(trigger);
}
```

Slika 13: Skripta AnimationScript.cs, SetTrigger funkcija

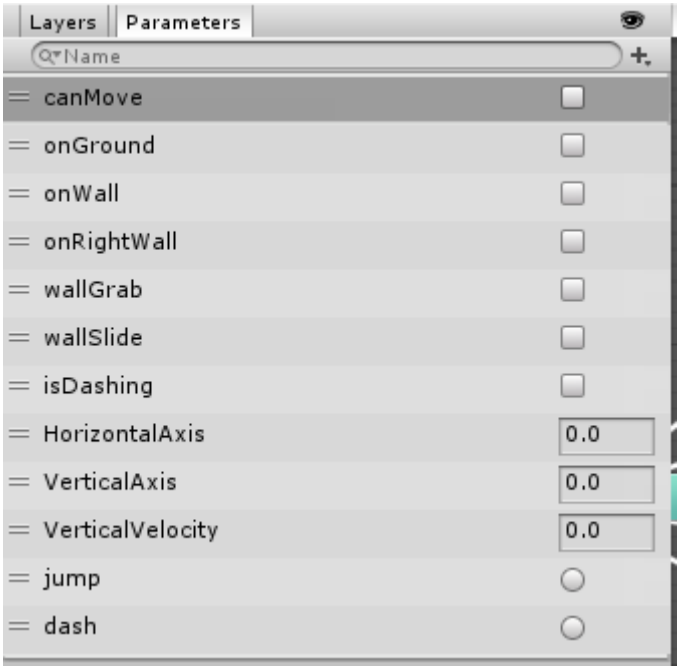
SetTrigger funkcija okida trigger parametar u animatoru.

```
public void Flip(int side)
{
    if (move.wallGrab || move.wallSlide)
    {
        if (side == -1 && sr.flipX)
            return;

        if (side == 1 && !sr.flipX)
        {
            return;
        }
    }
```

Slika 14: Skripta AnimationScript.cs, Flip funkcija

```
        bool state = (side == 1) ? false : true;
        sr.flipX = state;
    }
}
```

Funkcija Flip prima int vrijednost, te ovisno o tome da li događaju wallGrab ili wallSlide iz Movement skripte rotira sprite lika ovisno o smjeru kretanja, Vrijednost 1 znači da je sprite lika rotiran u desno, a -1 u lijevo.



Slika 15: Prikaz parametara animacija, Izvor: Unity Editor

5. MEHANIKE IGRE

U igri Transcendence jedan od najbitnijih elemenata igre jest kretanje lika. Pošto je igri fokus na glatko i brzo kretanje, izbjegavanje zamki i skakanje po zidovima i platformama iznimno je bitno da to radi glatko. Skripta Movement.cs jest implementacija kretanja lika.

Unutar klase su deklarirane sve potrebne varijable. Sve javne varijable (eng. public) se mogu izmjenjivati u samome editoru.

```
public class Movement : MonoBehaviour
{
    private Collision coll;
    [HideInInspector]
    public Rigidbody2D rb;
    private AnimationScript anim;

    [Space]
    [Header("Stats")]
    public float speed = 10;
    public float jumpForce = 50;
    public float slideSpeed = 5;
    public float wallJumpLerp = 10;
    public float dashSpeed = 20;

    [Space]
    [Header("Booleans")]
    public bool canMove;
    public bool wallGrab;
    public bool wallJumped;
    public bool wallSlide;
    public bool isDashing;
```

Slika 16: Skripta Movement.cs, varijable prvi dio

```
[Space]
private bool groundTouch;
private bool hasDashed;

public int side = 1;

[Space]
[Header("Polish")]
public ParticleSystem dashParticle;
public ParticleSystem jumpParticle;
public ParticleSystem wallJumpParticle;
public ParticleSystem slideParticle;
```

Slika 17: Skripta Movement.cs, varijable drugi dio

Unutar Start funkcije se dohvaćaju potrebne komponente.

```
void Start()
{
    coll = GetComponent<Collision>();
    rb = GetComponent<Rigidbody2D>();
    anim = GetComponentInChildren<AnimationScript>();
}
```

Slika 18: Skripta Movement.cs, Start funkcija

Unutar Update funkcije se provjeravaju uvjeti za sva potrebna kretanja te se izvršavaju i pozivaju odgovarajuće animacije na temelju inputa od strane igrača.

```
void Update()
{
    float x = Input.GetAxis("Horizontal");
    float y = Input.GetAxis("Vertical");
    float xRaw = Input.GetAxisRaw("Horizontal");
    float yRaw = Input.GetAxisRaw("Vertical");
    Vector2 dir = new Vector2(x, y);
    Walk(dir);
    anim.SetHorizontalMovement(x, y, rb.velocity.y);
```

Slika 19: Skripta Movement.cs, Update funcija, inputi kretanja

U prethodnome bloku koda, Update funkcija prima inpute od strane igrača, te ovisno o smjeru kretanja šalju se float vrijednosti u parametre animatora koji onda okida odgovarajuće animacije. Također se rotira slika lika ovisno o smjeru kretanja.

Inputi koje funkcija prima ovise o osima kretanja, odnosno po horizontalnoj osi (x osi), ili po vertikalnoj osi (y osi), prema tome unosi mogu biti tipke W, A, S, D ili strelice na tipkovnici.

```
if (coll.onWall && Input.GetButton("Fire3") && canMove)
{
    if (side != coll.wallSide)
        anim.Flip(side * -1);

    wallGrab = true;
    wallSlide = false;
}
```

Slika 20: Skripta Movement.cs, Update funkcija, provjera za držanje za zid

U prethodnome bloku koda provjerava se uvjet da li je lik u koliziji sa zidom, da li se može kretati i da li je pritisnut gumb naziva Fire3, odnosno lijevi Shift na tipkovnici.

```
if (Input.GetButtonUp("Fire3") || !coll.onWall || !canMove)
{
    wallGrab = false;
    wallSlide = false;
}
```

Slika 21: Skripta Movement.cs, Update funkcija, provjera za prestanak držanja za zid

U prethodnome bloku koda provjerava se uvjet da li je dignut gumb Fire3, odnosno da više nije pritisnut, ili da li lik više nije u koliziji sa zidom ili da se lik ne može kretati. Ako je jedan od tih uvjeta istinit onda se prekidaju wallGrab i wallSlide radnje.

```
if (coll.onGround && !isDashing)
{
    wallJumped = false;
    GetComponent<BetterJumping>().enabled = true;
}
```

Slika 22: Skripta Movement.cs, Update funkcija, provjera za omogućavanje BetterJumping komponente

U prethodnome bloku koda provjeravaju se uvjeti da li je lik na tlu odnosno u koliziji s tlom i da nije u isDashing stanju. Ako je uvjet istinit wallJumped vrijednost se postavlja na false i uključuje se BetterJumping komponenta.

```
if (wallGrab && !isDashing)
{
    rb.gravityScale = 0;
    if (x > .2f || x < -.2f)
        rb.velocity = new Vector2(rb.velocity.x, 0);

    float speedModifier = y > 0 ? .5f : 1;
    rb.velocity = new Vector2(rb.velocity.x, y * (speed * speedModifier));
}
else
{
    rb.gravityScale = 3;
}
```

Slika 23: Skripta Movement.cs, Update funkcija, provjera za penjanje i spuštanje po zidu

U prethodnome bloku koda provjerava se da li je wallGrab vrijednost istinita i da lik nije u isDashing stanju. Ako su ti uvjeti istiniti onda se vrijednost utjecaja gravitacije na tijelo lika postavlja na 0.

Ako je x veći od 0.2 ili manji od -0.2 tada vrijednost velocity poprima novu Vector2 vrijednost. Ako je y vrijednost veća od 0, tada speedModifier vrijednost se postavlja na 0.5 u suprotnome je 1, te se množi s velocity vrijednošću i rezultat postaje nova velocity vrijednost.

Odnosno, dok god je lik u wallGrab i igrač daje vertikalni input može se penjati odnosno spuštati po zidu. Penjanje je sporije od spuštanja.

```
if (coll.onWall && !coll.onGround)
    {
        if (x != 0 && !wallGrab)
        {
            wallSlide = true;
            WallSlide();
        }
    }
```

Slika 24: Skripta Movement.cs, Update funkcija, provjera za pozivanje WallSlide funkcije

U prethodnome bloku koda ako je lik u koliziji sa zidom i nije u na tlu, te zatim ako je x različit od 0 i lik nije u wallGrab stanju, varijabla wallSlide se postavlja na istinu (eng. true) i poziva se WallSlide funkcija.

```
if (!coll.onWall || coll.onGround)
        wallSlide = false;

    if (Input.GetButtonDown("Jump"))
    {
        anim.SetTrigger("jump");

        if (coll.onGround)
            Jump(Vector2.up, false);

        if (coll.onWall && !coll.onGround)
            WallJump();
    }
```

Slika 25: Skripta Movement.cs, Update funkcija, provjera za skakanje

U prethodnome bloku koda ako su uvjeti zadovoljeni izvodi se akcija skoka prilikom pritiska na tipku Space.

```
if (Input.GetButtonDown("Fire1") && !hasDashed)
    {
        if (xRaw != 0 || yRaw != 0)
            Dash(xRaw, yRaw);
    }
```

Slika 26: Skripta Movement.cs, Update funkcija, provjera za pozivanje Dash funkcije

Prethodni blok koda okida Dash funkciju ako su uvjeti zadovoljeni, odnosno ako je pritisnut input Fire1 (Control tipka na tipkovnici ili lijevi klik miša) i ako je hasDashed varijabla postavljena na laž (eng. false).

```
if (coll.onGround && !groundTouch)
    {
        GroundTouch();
        groundTouch = true;
    }
```

Slika 27: Skripta Movement.cs, Update funkcija, provjera za funkciju GroundTouch

Prethodni blok koda provjerava da li se događa kolizija s tlom i da li je varijabla groundTouch postavljena na laž (eng. false). Ako su uvjeti zadovoljeni, poziva se GroundTouch funkcija i varijabla groundTouch se postavlja na istinu (eng. true).

```
    if (!coll.onGround && groundTouch)
    {
        groundTouch = false;
    }
```

Slika 28: Skripta Movement.cs, Update funkcija, provjera za groundTouch vrijednost

Prethodni blok koda postavlja groundTouch vrijednost na laž ako su uvjeti zadovoljeni.

```
WallParticle(y);
    if (wallGrab || wallSlide || !canMove)
        return;
```

Slika 29: Skripta Movement.cs, Update funkcija, pozivanje sustava emitiranja čestica za WallParticle funkciju

```
    if (x > 0)
    {
        side = 1;
        anim.Flip(side);
    }
    if (x < 0)
    {
        side = -1;
        anim.Flip(side);
    }
}
```

Slika 30: Skripta Movement.cs, Update funkcija, rotiranje slike lika ovisno o smjeru kretanja lika

Prethodni blok koda provjerava vrijednosti x varijable. Ako je x veći od nula, odnosno smjer kretanja lika jest udesno, što znači da je horizontalna vrijednost x veća od 0, tada se vrijednost side, odnosno strana, postavlja na 1, te se poziva funkcija flip u animatoru koji onda rotira sliku lika u odgovarajućem smjeru, u ovome slučaju u desno.

U suprotnome, ako je x manji od 0, onda je slika rotirana u lijevo.

```
void GroundTouch()
{
    hasDashed = false;
    isDashing = false;

    side = anim.sr.flipX ? -1 : 1;
    jumpParticle.Play();
}
```

Slika 31: Skripta Movement.cs, GroundTouch funkcija

Funkcija GroundTouch služi za provjeru kontakta s tlom, mijenjanje smjera gledanja slike igrača ovisno o smjeru kretanja i za emitiranje čestica kada igrač skoči.

```
  private void Dash(float x, float y)
  {
      Camera.main.transform.DOComplete();
      Camera.main.transform.DOShakePosition(.2f, .5f, 14, 90, false, true);
FindObjectOfType<RippleEffect>().Emit(Camera.main.WorldToViewportPoint(transform.p
osition));

      hasDashed = true;
      anim.SetTrigger("dash");
      rb.velocity = Vector2.zero;
      Vector2 dir = new Vector2(x, y);
      rb.velocity += dir.normalized * dashSpeed;
      StartCoroutine(DashWait());
  }
```

Slika 32: Skripta Movement.cs, Dash funkcija

Dash funkcija omogućava igraču nagli polet kroz zrak. Detektira kada igrač unese input, zatim šalje signal animatoru da odigra odgovarajuću animaciju, te pronalazi objekt zvan RippleEffect koji je skripta na kameri, te je opisana dalje u radu.

Prilikom svake uporabe dasha također se i zatrese kamera za dodatni efekt. Nakon što se izvede dash poziva se korutina DashWait() koja je odgovorna za emitiranje čestica za efekt dasha i postavljanja određenih parametara.

```
IEnumerator DashWait()
  {
      FindObjectOfType<GhostTrail>().ShowGhost();
      StartCoroutine(GroundDash());
      DOVirtual.Float(14, 0, .8f, RigidbodyDrag);

      dashParticle.Play();
      rb.gravityScale = 0;
      GetComponent<BetterJumping>().enabled = false;
```

Slika 33: Skripta Movement.cs, korutina DashWait, prvi dio

```
    wallJumped = true;
    isDashing = true;
    yield return new WaitForSeconds(.3f);

    dashParticle.Stop();
    rb.gravityScale = 3;
    GetComponent<BetterJumping>().enabled = true;
    wallJumped = false;
    isDashing = false;
}
```

Slika 34: Skripta Movement.cs, korutina DashWait, drugi dio

Prethodni blok koda jest DashWait korutina (eng. Coroutine) koja se pokreće unutar Dash funkcije. Kada se pokrene prvo traži objekt na sceni naziva Ghost trail i okida funkciju ShowGhost. Zatim starta korutinu GroundDash i mijenja vrijednosti RigidbodyDrag varijable.

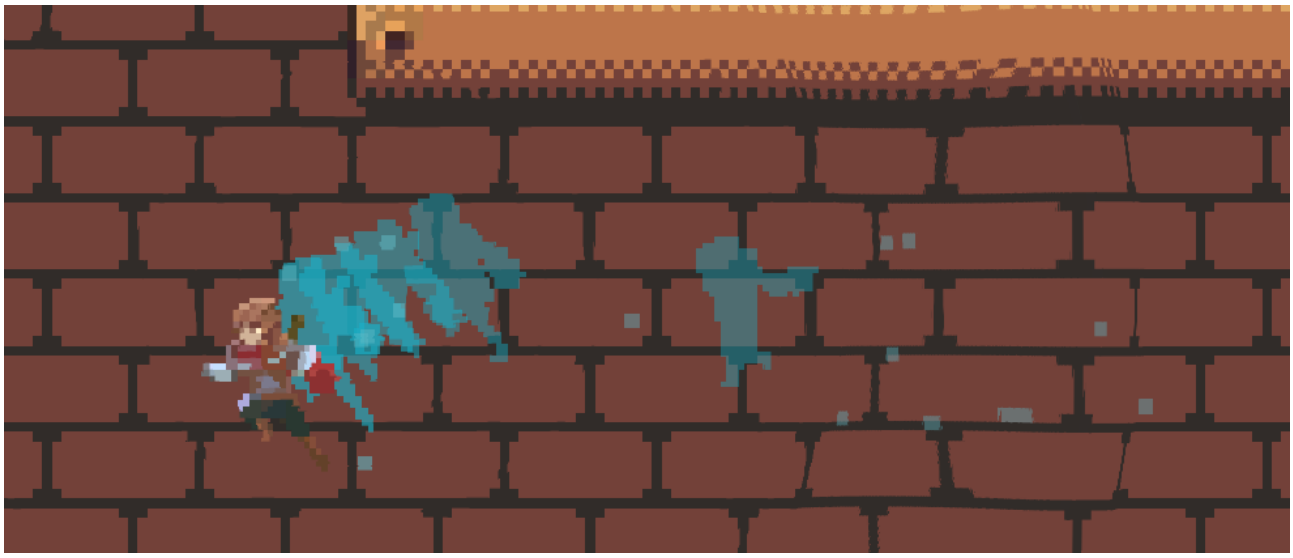
Zatim se emitiraju čestice vezane za Dash akciju, utjecaj gravitacije se postavlja na 0, onemogućuje se komponenta BetterJumping i postavljaju se vrijednost wallJumped i isDashing varijabli na istinu.

Nakon toga čeka 0.3 sekunde, te onda prekida emitiranje čestica, postavlja utjecaj gravitacije natrag na 3, omogućava komponentu BetterJumping i postavlja vrijednosti wallJumped i isDashing varijabli na laž.

```
IEnumerator GroundDash()
{
    yield return new WaitForSeconds(.15f);
    if (coll.onGround)
        hasDashed = false;
}
```

Slika 35: Skripta Movement.cs, GroundDash korutina

GroundDash korutina se poziva unutar DashWait korutine. Pokreće se brojač od 0.15 sekundi te ako je uvjet kolizija s tlom istinit isDashed varijabla se postavlja na laž.



Slika 36: Snimak igre, primjer naglog poleta, odnosno Dash funkcije, izvor; projekt

```
private void WallJump()
  {
    if ((side == 1 && coll.onRightWall) || side == -1 && !coll.onRightWall)
    {
      side *= -1;
      anim.Flip(side);
    }

    StopCoroutine(DisableMovement(0));
    StartCoroutine(DisableMovement(.1f));

    Vector2 wallDir = coll.onRightWall ? Vector2.left : Vector2.right;

    Jump((Vector2.up / 1.5f + wallDir / 1.5f), true);
    wallJumped = true;
  }
```

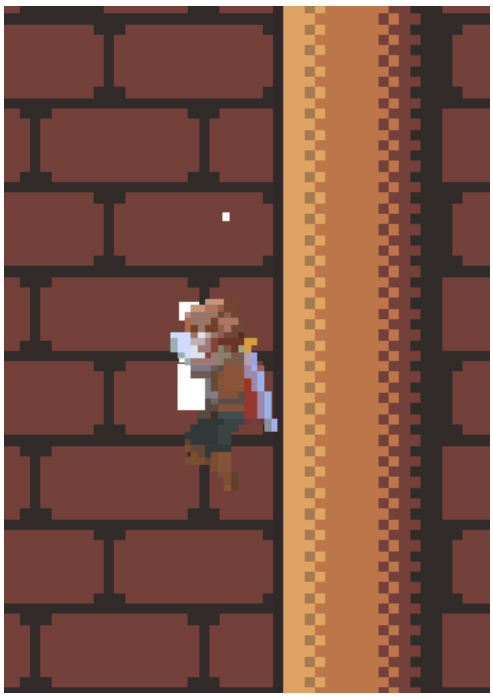
Slika 37: Skripta Movement.cs, WallJump funkcija,

WallJump funkcija omogućava skakanje sa zida na zid. Pritom mijenja rotaciju slike suprotno od zida s kojim je lik u koliziji, poziva DisableMovement korutinu, poziva Jump funkciju kojoj mijenja standardne vrijednosti i postavlja wallJumped varijablu na istinu.

```
private void WallSlide()
{
    if (coll.wallSide != side)
        anim.Flip(side * -1);

    if (!canMove)
        return;
    bool pushingWall = false;
    if ((rb.velocity.x > 0 && coll.onRightWall) || (rb.velocity.x < 0 && coll.onLeftWall))
    {
        pushingWall = true;
    }
    float push = pushingWall ? 0 : rb.velocity.x;
    rb.velocity = new Vector2(push, -slideSpeed);
}
```

Slika 38: Skripta Movement.cs, WallSlide funkcija

Funkcija WallSlide omogućava klizanje lika niz zid prilikom kontakta sa zidom.



Slika 39: Snimak igre, prikaz klizanja zidom, odnosno WallSlide funkcije

```
private void Walk(Vector2 dir)
{
    if (!canMove)
        return;
    if (wallGrab)
        return;

    if (!wallJumped) {
        rb.velocity = new Vector2(dir.x * speed, rb.velocity.y);
    }
    else  {
         rb.velocity = Vector2.Lerp(rb.velocity, (new Vector2(dir.x * speed, rb.velocity.y)),
wallJumpLerp * Time.deltaTime);
    }
}
```

Slika 40: Skripta Movement.cs, Walk funkcija

Funkcija Walk omogućava kretanje lika, odnosno hodanje.

```
private void Jump(Vector2 dir, bool wall)
{
    slideParticle.transform.parent.localScale = new Vector3(ParticleSide(), 1, 1);
    ParticleSystem particle = wall ? wallJumpParticle : jumpParticle;
    rb.velocity = new Vector2(rb.velocity.x, 0);
    rb.velocity += dir * jumpForce;
    particle.Play();
}
```

Slika 41: Skripta Movement.cs, Jump funkcija

Funkcija Jump omogućava skakanje lika.

```
IEnumerator DisableMovement(float time)
{
    canMove = false;
    yield return new WaitForSeconds(time);
    canMove = true;
}
```
Slika 42: Skripta Movement.cs, korutina DisableMovement

DisableMovement korutina blokira mogućnost kretanja lika kada se pozove.

```
void RigidbodyDrag(float x)
 {
    rb.drag = x;
 }
```
Slika 43: Skripta Movement.cs, RigidbodyDrag funkcija

RigidbodyDrag funkcija omogućava podešavanje drag vrijednosti.

```
void WallParticle(float vertical)
{
    var main = slideParticle.main;
    if (wallSlide || (wallGrab && vertical < 0))
    {
        slideParticle.transform.parent.localScale = new Vector3(ParticleSide(), 1, 1);
        main.startColor = Color.white;
    }
    else
    {
        main.startColor = Color.clear;
    }
}
```
Slika 44: Skripta Movement.cs, WallParticle funkcija

```
    int ParticleSide()
    {
        int particleSide = coll.onRightWall ? 1 : -1;
        return particleSide;
    }
}
```

Slika 45: Skripta Movement.cs, ParticleSide funkcija

WallParticle funkcija kontrolira emitiranje čestica kada lik klizi po zidu. ParticleSide funkcija kontrolira na kojoj će se strani lika prilikom klizanja niz zid emitirati sustav čestica.

Skripta Collision.cs se brine za detekciju kolizije sa slojem ground kako bi se animacije pravilno okidale ovisno o terenu s kojim je lik u kontaktu.

```
public class Collision : MonoBehaviour
{

    [Header("Layers")]
    public LayerMask groundLayer;

    [Space]
    public bool onGround;
    public bool onWall;
    public bool onRightWall;
    public bool onLeftWall;
    public int wallSide;

    [Space]
    [Header("Collision")]
    public float collisionRadius = 0.25f;
    public Vector2 bottomOffset, rightOffset, leftOffset;
    private Color debugCollisionColor = Color.red;
```

Slika 46: Skripta Collision.cs, varijable

```
void Update()
{
 onGround = Physics2D.OverlapCircle((Vector2)transform.position + bottomOffset,
      collisionRadius, groundLayer);
 onWall = Physics2D.OverlapCircle((Vector2)transform.position + rightOffset,
      collisionRadius, groundLayer)
|| Physics2D.OverlapCircle((Vector2)transform.position + leftOffset, collisionRadius,
      groundLayer);

onRightWall = Physics2D.OverlapCircle((Vector2)transform.position + rightOffset,
      collisionRadius, groundLayer);
onLeftWall = Physics2D.OverlapCircle((Vector2)transform.position + leftOffset,
      collisionRadius, groundLayer);
   wallSide = onRightWall ? -1 : 1;
}
```

Slika 47: Skripta Collision.cs, Update funkcija

Unutar Update funkcije se provjerava na temelju krugova kolizije da li dolazi do presjeka između tih krugova i tla, odnosno zidova.

```
void OnDrawGizmos()
{
   Gizmos.color = Color.red;
   var positions = new Vector2[] { bottomOffset, rightOffset, leftOffset };

   Gizmos.DrawWireSphere((Vector2)transform.position + bottomOffset,
      collisionRadius);
   Gizmos.DrawWireSphere((Vector2)transform.position + rightOffset, collisionRadius);
   Gizmos.DrawWireSphere((Vector2)transform.position + leftOffset, collisionRadius);
}
}
```

Slika 48: Skripta Collision.cs, OnDrawGizmos funkcija

OnDrawGizmos funkcija unutar editora iscrtava krugove kolizije unutar editora za pomoć pri namještanju promjera kolizije.

Slika 49: Snimak igre, sustav detekcije kolizije, izvor; projekt

BetterJumping.cs je skripta koja omogućuje bolju kontrolu nad varijablama skakanja. Ako se prilikom akcije skoka nastavi držati tipka Spacebar, skok će biti viši nego da se pritisne i odmah pusti.

```
public class BetterJumping : MonoBehaviour
{
    private Rigidbody2D rb;
    public float fallMultiplier = 2.5f;
    public float lowJumpMultiplier = 2f;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
    }
```

Slika 50: Skripta BetterJumping.cs, varijable i Start funkcija

```
    void Update()
    {
        if(rb.velocity.y < 0)
        {
                rb.velocity += Vector2.up * Physics2D.gravity.y * (fallMultiplier - 1) *
Time.deltaTime;
        }
        else if(rb.velocity.y > 0 && !Input.GetButton("Jump"))
        {
                rb.velocity += Vector2.up * Physics2D.gravity.y * (lowJumpMultiplier - 1) *
Time.deltaTime;
        }
    }
}
```

Slika 51: Skripta BetterJumping.cs, Update funkcija

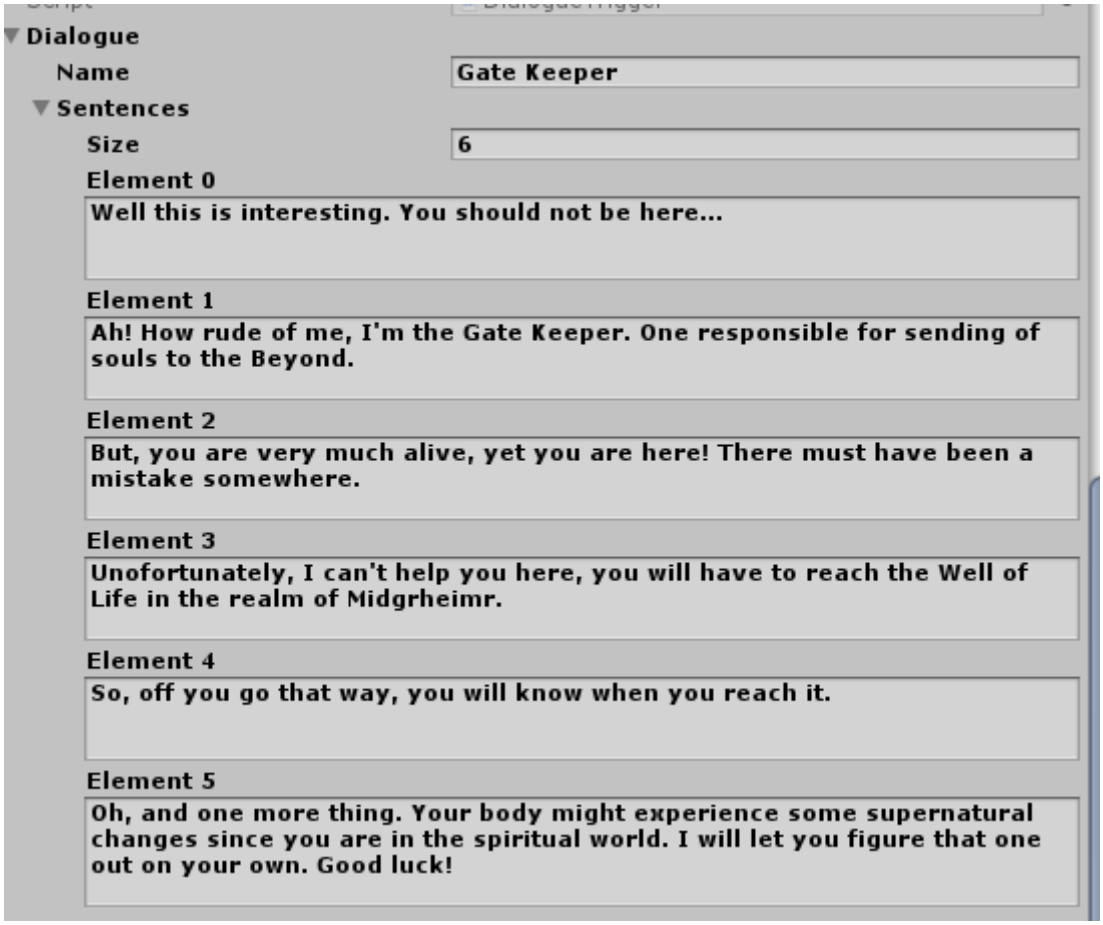
5.1 DIJALOG

Na svakoj razini igre, igrač upoznaje neigrive, NPC (eng.non playable character) likove s kojima vodi razgovore.

Dialogue.cs je skripta koja se koristi samo kao objekt koji se predaje Dialogue Manageru svaki put kada se započinje novi dijalog. Ova skripta sadržava sve informacije o dijalogu. Kako bi se skripta pojavila u editora da se može u nju unositi dijalog potrebno ju je serijalizirati.

```
[System.Serializable]
public class Dialogue
{
    public string name;
    [TextArea(3, 10)]
    public string[] sentences;
}
```

Slika 52: Skripta Dialogue.cs



Slika 53: Snimak igre, primjer unošenja dijalogu u editoru, izvor; projekt

DialogueManager skripta je odgovorna za kontroliranje toka dijaloga. Za pohranjivanje svih rečenica u dijalogu koristi se red. Unutar Start funkcije inicijalizira se novi red.

```
public class DialogueManager : MonoBehaviour
{
    public Text nameText;
    public Text dialogueText;
    public Animator animator;
    private Queue<string> sentences;

    void Start()
    {
        sentences = new Queue<string>();
    }
```

Slika 54: Skripta DialogueManager.cs, varijable i Start funkcija

StartDialogue funkcija prvo briše sve rečenice iz prijašnjih dijaloga, zatim iterira kroz sve rečenice u sentences polju u Dialogue.cs skripti, te ih stavlja u red. Zatim prikazuje prvu na ekran, odnosno unutar sučelja predviđenog za to.

```
public void StartDialogue(Dialogue dialogue)
{
    animator.SetBool("IsOpen", true);
    nameText.text = dialogue.name;
    sentences.Clear();

    foreach (string sentence in dialogue.sentences)
    {
        sentences.Enqueue(sentence);
    }
    DisplayNextSentence();
}
```

Slika 55: Skripta DialogueManager.cs, StartDialogue funkcija

Korutina TypeSentence ispisuje rečenice slovo po slovo kao da su animirana.

```
IEnumerator TypeSentence (string sentence)
{
    dialogueText.text = "";

    foreach (char letter in sentence.ToCharArray())
    {
        dialogueText.text += letter;
        yield return null;
    }
}
```

Slika 56: Skripta DialogueManager.cs, korutina TypeSentence

Funkcija DisplayNextSentence ispisuje sljedeću rečenicu dijaloga u redu. Dok EndDialogue funkcija zatvara sučelje dijaloga kada se pozove unutar DisplayNextSentence funkcija, ako broj rečenica za ispisati je jednak nuli.

```
    public void DisplayNextSentence()
    {
        if (sentences.Count == 0)
        {
            EndDialogue();
            return;
        }

        string sentence = sentences.Dequeue();
        StopAllCoroutines();
        StartCoroutine(TypeSentence(sentence));
    }

    void EndDialogue()
    {
        animator.SetBool("IsOpen", false);
    }
}
```

Slika 57: Skripta DialogueManager.cs, funkcije DisplayNextSentence i EndDialogue

StartDialogue funkcija započinje dijalog kada lik uđe u koliziju s objektom predviđenim za startanje dijaloga.

```
public class StartDialogue : MonoBehaviour
{
    public DialogueTrigger dialogueTrigger;

    private void OnTriggerEnter2D(Collider2D collision){
        dialogueTrigger.TriggerDialogue();
        Debug.Log("Dialogue Started");
    }
}
```

Slika 58: Skripta DialogueManager.cs, StartDialogue funkcija

Slika 59: Snimak igre, primjer dijaloga unutar igre, izvor; projekt

DialogueTrigger funkcija pronalazi komponentu tipa DialogueManager i okida funkciju StartDialogue.

```
public class DialogueTrigger : MonoBehaviour
{
    public Dialogue dialogue;

    public void TriggerDialogue()
    {
        FindObjectOfType<DialogueManager>().StartDialogue(dialogue);
    }
}
```

Slika 60: Skripta DialogueTrigger

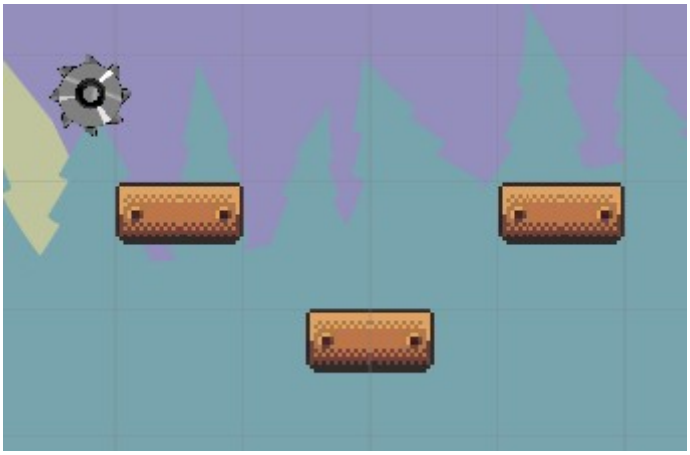
5.2 Zamke

Kroz igru igrač se susreće s različitim zamkama kao što su; blokovi koji ispaljuju projektile, pokretne zamke poput rotirajuće pile, platformi koje nestaju pri kontaktu i smrtonosni teren.

```
public class CircularRotation : MonoBehaviour
{
    public float RotateSpeed = 5f;
    public float Radius = 0.1f;
    private Vector2 centre;
    private float angle;

    void Start()
    {
        centre = transform.position;
    }
```

Slika 61: Skripta CircularRotation.cs, varijable i Start funkcija

CircularRotation skripta pomiče objekt na koje je zakačena u kružnoj putanji. Skripta se može primijeniti i na platformama, pa su potrebne OnCollisionEnter2D i OnCollisionExit2D funkcije u kojima se lik postavlja kao dijete platforme na kojoj se nalazi kako bi se mogao i dalje normalno kretati po njoj, u protivnome bi samo skliznuo s platforme.



Slika 62: Snimak igre, primjer zamke rotirajuće pile, izvor; projekt

```
void Update()
  {
     angle += RotateSpeed * Time.deltaTime;

     var offset = new Vector2(Mathf.Sin(angle), Mathf.Cos(angle)) * Radius;
     transform.position = centre + offset;
  }

  private void OnCollisionEnter2D(Collision2D collision)
  {
     if (collision.gameObject.tag == "Player")
     {
        collision.collider.transform.SetParent(transform);
     }
  }

  private void OnCollisionExit2D(Collision2D collision)
  {
     if (collision.gameObject.tag == "Player")
     {
        collision.collider.transform.SetParent(null);
     }
  }
}
```

Slika 63: Skripta CircularRotation.cs, Update funkcija i funckije za ulaz i izlaz iz kolizije

DamageZone skripta se stavlja na objekte koji štete igraču. Igrač ima skrivenu vrijednost života koja se smanjuje pri kontaktu sa zamkama ili štetnim terenom. Igrač ima samo jedan život i ako umre vraća se na početak trenutne razine na kojemu se nalazi.

```
public class DamageZone : MonoBehaviour
{
    private void OnTriggerStay2D(Collider2D other)
    {
        PlayerController controller = other.GetComponent<PlayerController>();

        if (controller != null)
        {
            controller.ChangeHealth(-1);
            Debug.Log("Health reduced");

            Destroy(gameObject);
            Destroy(other.gameObject);
            Debug.Log("Destroyed");
        }
    }
}
```

Slika 64: DamageZone.cs skripta

Jedne od zamki na koje igrač nailazi unutar igre su sklopive platforme. Za kontrolu tih platformi odgovorna je PlatformFold skripta.

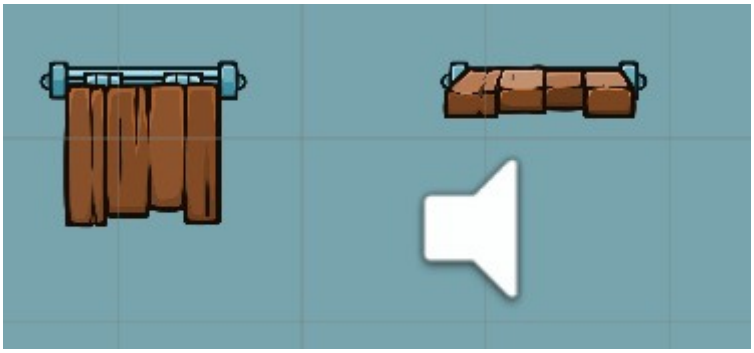
Kada igrač stane na platformu okida se brojač od 3 sekunde, kada brojač dođe do 0 onemogućuje se kolizija na objektu i pokreće se animacija sklapanja platforma. Ako se igrač u tome trenu nalazi na platformi pasti će s platforme. Jednom kada igrač izađe iz kolizije s platformom pokreče se brojač za restartanje platforme koja se na kraju brojača vraća u prvobitno stanje

```
public class PlatformFold : MonoBehaviour
{
    [SerializeField] float cooldown = 5f;
    [SerializeField] float timeToFold = 3f;
    Animator animator;
    float cooldownTimer;
    float foldTimer;
    bool startCooldownTimer;
    bool startfoldTimer;

    void Start()
    {
        animator = gameObject.GetComponent<Animator>();
        cooldownTimer = cooldown;
        foldTimer = timeToFold;
    }

    private void Update()
    {
        if (startfoldTimer == true)
        {
            foldTimer -= Time.deltaTime;
            if (foldTimer < 0)
            {
                gameObject.GetComponent<BoxCollider2D>().enabled = false;
                animator.SetTrigger("SteppedOn");
                foldTimer = timeToFold;
            }
        }
```

Slika 65: Skripta PlatformFold.cs, varijble, Start i Update funkcija

```
if (gameObject.GetComponent<BoxCollider2D>().enabled == false)
    {
        startfoldTimer = false;
        cooldownTimer -= Time.deltaTime;

        if (cooldownTimer < 0)
        {
            gameObject.GetComponent<BoxCollider2D>().enabled = true;
            animator.SetTrigger("Reset");
            cooldownTimer = cooldown;
        }
    }
  }
  private void OnCollisionEnter2D(Collision2D collision)
  {
     startfoldTimer = true;
  }
}
```

Slika 66: Skripta PlatformFold.cs, Update funckija i provjera kolizije



Slika 67: Snimak igre, primjer sklopive platforme, izvor; projekt

Kroz igru nailazi na puno prepreka koje ispaljuju projektile koje igrač mora izbjeći. Jedna od skripti odgovornih za logiku tih zamki jest TrapShootingLogicDown. Skripta ima brojač koji svaki put kada dođe do 0 instancira projektil koje putuje u zadanome smjeru.

```
public class TrapShootingLogicDown : MonoBehaviour
{
    public float shootingInterval = 3f;
    public float shootingSpeed = 2f;
    public float timeToDestroy = 2f;
    public GameObject projectilePrefab;
    private float shootingTimer;

    void Start()
    {
        shootingTimer = shootingInterval;
    }
    void Update()
    {
        shootingTimer -= Time.deltaTime;

        if (shootingTimer <= 0f)
        {
            shootingTimer = shootingInterval;

            GameObject projectileInstance = Instantiate(projectilePrefab);
            projectileInstance.transform.position = transform.position;

                projectileInstance.GetComponent<Rigidbody2D>().velocity = new Vector2(0, -
shootingSpeed);
            Destroy(projectileInstance, timeToDestroy);
        }
    }
}
```

Slika 68: Skripta TrapShootingLogicDown.cs

5.3 Mogućnost mijenjanja forme lika

Igrač ima mogućnost mijenjanja formi lika. Igrač bira između forme čovjeka i mačke.

Skripta odgovorna za to jest CharacterSwitch.

```
using UnityEngine;

public class CharacterSwitch : MonoBehaviour
{
    public GameObject avatar1;
    public GameObject avatar2;

    int whichAvatarIsOn = 1;
```

Slika 69: Skripta CharacterSwitch.cs, varijable

Slika 70: Snimak igre, druga forma glavnoga lika, izvor; projekt

```
void Start() {
    avatar1.gameObject.SetActive(true);
    avatar2.gameObject.SetActive(false);
  }

  void LateUpdate() {
    var offset = new Vector3(0, 0.5f, 0);
    SwitchAvatar();
    if (whichAvatarIsOn == 2)
    {
        avatar1.transform.position = avatar2.transform.position;
    }
    else {
        avatar2.transform.position = avatar1.transform.position + offset;
    }
  }

  public void SwitchAvatar()  {
    if (Input.GetKeyDown(KeyCode.Alpha2))
    {
        whichAvatarIsOn = 2;
        avatar1.gameObject.SetActive(false);
        avatar2.gameObject.SetActive(true);
    }
    else if (Input.GetKeyDown(KeyCode.Alpha1)){
        whichAvatarIsOn = 1;
        avatar1.gameObject.SetActive(true);
        avatar2.gameObject.SetActive(false);
    }
  }
}
```

Slika 71: Skripta CharacterSwitch.cs, funckije Start, LateUpdate i SwitchAvatar

5.4 Parallax efekt

Pomicanje paralaksom je tehnika u računalnoj grafici u kojoj se pozadinske slike sporije provlače ispred kamere nego prednje slike, stvarajući iluziju dubine u 2D sceni.

Skripta koja kontrolira efekt jest Parallax skripta. Unutar editora se popunjava polje sa slikama na koje se želi primijeniti efekt. Unutar Awake metode se postavlja referenca na glavnu kameru.

U Start metodi se pohranjuje prošli okvir (eng. frame) kamere i dodjeljuju se pripadajuće skale paralaksa.

Unutar Update metode, za svaku sliku, paralaks je suprotan od smjera kretanja kamere zbog prošlog okvira kamere pomnoženog sa skalom paralaksa. Postavlja se ciljana x pozicija koja je trenutna x pozicija plus paralaks. Zatim se kreira ciljana pozicija koja je trenutna pozicija slike s njenom ciljanom x pozicijom. Zatim se izmjenjuju trenutna i ciljana pozicija pomoću funkcije linearne interpolacije (eng. Lerp), te se naposljetku postavlja prethodna pozicija kamere na trenutnu poziciju kamere na kraju okvira.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Parallax : MonoBehaviour
{
    public Transform[] backgrounds;
    private float[] parallaxScales;
    public float smoothing = 0.2f;
    private Transform cam;
    private Vector3 previousCamPosition;

    private void Awake()
    {
        cam = Camera.main.transform;
    }
```

Slika 72: Skripta Parallax.cs, varijable i Awake funkcija

```
    void Start()
    {
        previousCamPosition = cam.position;
        parallaxScales = new float[backgrounds.Length];

        for (int i = 0; i < backgrounds.Length; i++)
        {
            parallaxScales[i] = backgrounds[i].position.z;
        }
    }

    void Update()
    {
        for (int i = 0; i < backgrounds.Length; i++)
        {
            float parallax = (previousCamPosition.x - cam.position.x) * parallaxScales[i];

            float backgroundTargetPositionX = backgrounds[i].position.x + parallax;

                Vector3 backgroundTargetPosition = new Vector3(backgroundTargetPositionX,
backgrounds[i].position.y, backgrounds[i].position.z);

                                backgrounds[i].position = Vector3.Lerp(backgrounds[i].position,
backgroundTargetPosition, smoothing * Time.deltaTime);
        }
        previousCamPosition = cam.position;
    }
}
```

Slika 73: Skripta Parallax.cs, Start i Update funkcije

6. SHADERI

Tijekom izrade crteža s računalima, u tom procesu se crta krug, zatim pravokutnik, linija, neke trokuti sve dok se ne sastavi željena sliku. Taj je proces vrlo sličan pisanju pisma ili knjige rukom - to je skup uputa koje obavljaju jedan zadatak za drugim (The Book of Shaders, 2019).

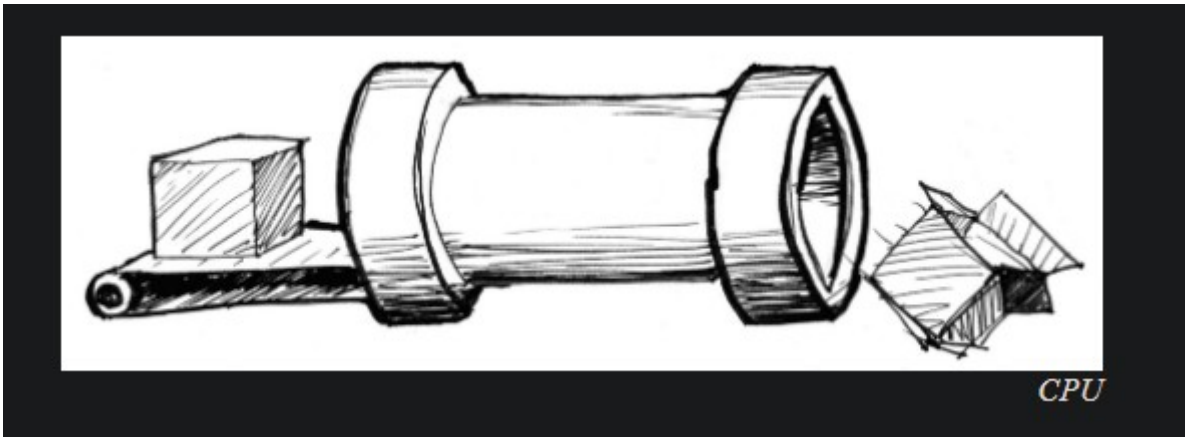
Shaderi su također skup uputa, ali se upute izvršavaju odjednom za svaki piksel na zaslonu. To znači da se kod koji se piše mora ponašati drugačije, ovisno o položaju piksela na zaslonu. Poput tiskarskog stroja, program će funkcionirati kao funkcija koja prima poziciju i vraća boju, a kada se kompilira pokrenut će se izuzetno brzo (The Book of Shaders, 2019).

Shaderi rade izuzetno brzo zbog paralelnog procesiranja. Paralelno procesiranje je vrsta računanja u kojoj se istovremeno izvode mnogi izračuni ili izvršavaju procesi. Veliki problemi često se mogu podijeliti na manje, koji se tada mogu riješiti (The Book of Shaders, 2019).

Procesor računala se može zamisliti kao veliku industrijsku cijev, a svaki zadatak kao nešto što prolazi kroz njega - poput tvorničke linije. Neke su zadaće veće od drugih, što znači da im je potrebno više vremena i energije za rješavanje. Kaže se da zahtijevaju veću procesorsku snagu (The Book of Shaders, 2019).

Zbog arhitekture računala poslovi su prisiljeni biti izvođeni u nizu; svaki posao mora biti završen jedan po jedan. Moderna računala obično imaju grupe od četiri procesora koji rade poput ovih cijevi, dovršavajući zadatke jedan za drugim kako bi stvari održavale glatko. Svaka cijev je također poznata kao nit (eng. thread) (The Book of Shaders, 2019).

Video igre i druge grafičke aplikacije zahtijevaju mnogo veću procesorsku snagu od ostalih programa. Zbog svog grafičkog sadržaja moraju raditi ogroman broj operacija piksela po pikselima. Svaki pojedinačni piksel na zaslonu treba izračunati, a u 3D igrama također treba izračunati geometriju i perspektive. Na slici 12 je prikazana metafora procesora kao industrijske cijevi (The Book of Shaders, 2019).



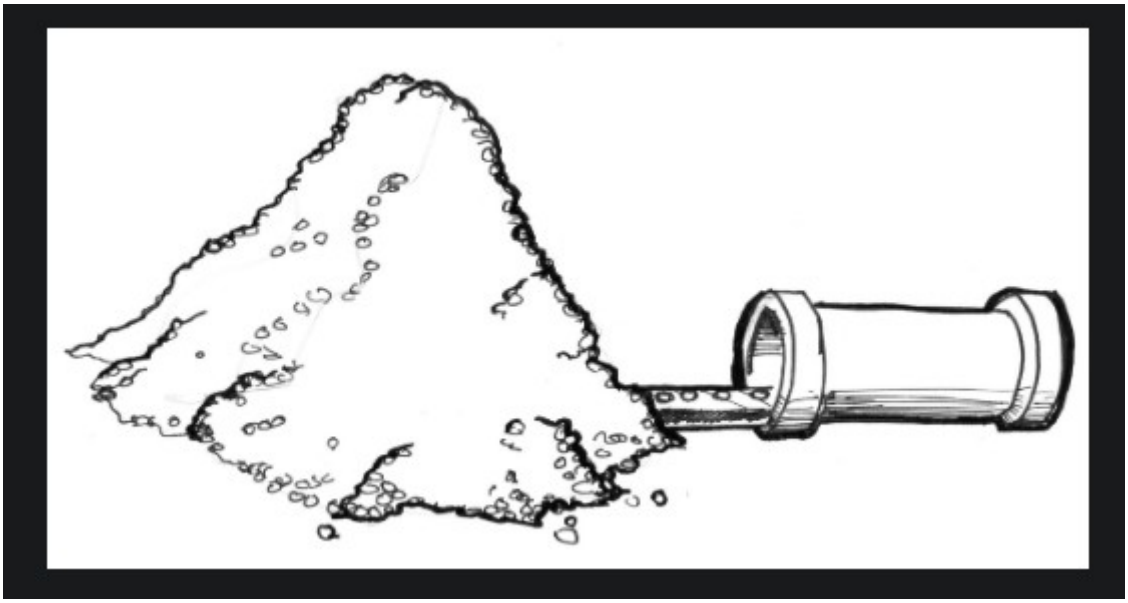
Slika 74: Metaforički prikaz procesora kao industrijske cijevi, izvor: The Book of Shaders, 2019

Svaki piksel na zaslonu predstavlja jednostavan zadatak. Pojedinačno svaki zadatak piksela nije problem za CPU, ali (i ovdje je problem) mali zadatak mora biti učinjen za svaki piksel na zaslonu! Metaforički prikaz na slici 13 (The Book of Shaders, 2019).

To znači da u starom 800x600 zaslonu 480.000 piksela treba obraditi po kadru što znači 14.400.000 izračuna u sekundi. To je dovoljno velik problem da preoptereti mikroprocesor. Na suvremenom 2880x1800 zaslonu mrežnice koji radi na 60 sličica u sekundi, izračun iznosi 311.040.000 izračuna u sekundi. (The Book of Shaders, 2019).

Tada paralelno procesiranje postaje dobro rješenje. Umjesto nekoliko velikih i moćnih mikroprocesora, ili cijevi, pametnije je imati puno malih sićušnih mikroprocesora koji rade paralelno u isto vrijeme. To je grafička procesorska jedinica (GPU). (The Book of Shaders, 2019).

Zamislite sićušne mikroprocesore kao stol cijevi i podatke svakog piksela kao lopticu za stolni tenis. 14.400.000 stolno teniskih loptica u sekundi može ometati gotovo svaku cijev. No, stol od 800x600 sićušnih cijevi koje primaju 30 valova od 480.000 piksela u sekundi može se rukovati glatko. To radi na višim razlučivostima ekrana - što više paralelnog hardvera se posjeduje, to je veći tok koji može upravljati. Metaforički prikaz na slici 14. (The Book of Shaders, 2019).



Slika 75: Metaforički prikaz reda zadataka koji čekaju na procesiranje, izvor: The Book of Shaders, 2019

Još jedna “super snaga” GPU-a su posebne matematičke funkcije koje se ubrzavaju putem hardvera, tako da kompliciranе matematičke operacije rješavaju izravno mikročipovi, a ne softver. To znači dodatne brze trigonometrijske i matrične operacije - jednako brzo kao i struja (The Book of Shaders, 2019).

Shaderi se pišu u programskome jeziku zvanome GLSL. GLSL je skraćenica za OpenGL Shading Language, koji je specifičan standard shader programa.

Ripple effect je efekt koji se izvodi svaki put kada se okine funkcija Dash(), odnosno kada igrač da input za taj pokret. Ono što shader radi jest simulacija efekta mreškanja valova kao kada se baci kamen u vodu, te od izvora udara kamena u vodi ide kružni val. Taj efekt se dogodi svaki put kada igrač koristi pokret Dash.

Slika 76: Snimak igre, primjer RippleEffect shadera, izvor; projekt

Skripta koja se stavlja kao komponenta na kameru, te podešavanje varijabli unutar editora omogućuje mijenjanje efekta po volji.

```
public class RippleEffect : MonoBehaviour
{
    public AnimationCurve waveform = new AnimationCurve(
        new Keyframe(0.00f, 0.50f, 0, 0),
        new Keyframe(0.05f, 1.00f, 0, 0),
        new Keyframe(0.15f, 0.10f, 0, 0),
        new Keyframe(0.25f, 0.80f, 0, 0),
        new Keyframe(0.35f, 0.30f, 0, 0),
        new Keyframe(0.45f, 0.60f, 0, 0),
        new Keyframe(0.55f, 0.40f, 0, 0),
        new Keyframe(0.65f, 0.55f, 0, 0),
        new Keyframe(0.75f, 0.46f, 0, 0),
        new Keyframe(0.85f, 0.52f, 0, 0),
        new Keyframe(0.99f, 0.50f, 0, 0)
    );
    [Range(0.01f, 1.0f)]
    public float refractionStrength = 0.5f;

    public Color reflectionColor = Color.gray;

    [Range(0.01f, 1.0f)]
    public float reflectionStrength = 0.7f;

    [Range(1.0f, 5.0f)]
    public float waveSpeed = 1.25f;

    [Range(0.0f, 2.0f)]
    public float dropInterval = 0.5f;

    [SerializeField, HideInInspector]
    Shader shader;
```

Slika 77: Skripta RippleEffect, varijable

```
class Droplet
  {
     Vector2 position;
     float time;

     public Droplet()
     {
        time = 1000;
     }

     public void Reset(Vector2 pos)
     {
       position = pos;
        time = 0;
     }

     public void Update()
     {
        time += Time.deltaTime * 2;
     }

     public Vector4 MakeShaderParameter(float aspect)
     {
        return new Vector4(position.x * aspect, position.y, time, 0);
     }
  }
```

Slika 78: Skripta RippleEffect.cs, Droplet klasa

```
    Droplet[] droplets;
    Texture2D gradTexture;
    Material material;
    float timer;
    int dropCount;

    void UpdateShaderParameters()
    {
        var c = GetComponent<Camera>();
        material.SetVector("_Drop1", droplets[0].MakeShaderParameter(c.aspect));
        material.SetVector("_Drop2", droplets[1].MakeShaderParameter(c.aspect));
        material.SetVector("_Drop3", droplets[2].MakeShaderParameter(c.aspect));
        material.SetColor("_Reflection", reflectionColor);
        material.SetVector("_Params1", new Vector4(c.aspect, 1, 1 / waveSpeed, 0));
        material.SetVector("_Params2", new Vector4(1, 1 / c.aspect, refractionStrength,
reflectionStrength));
    }
    void Awake() {
        droplets = new Droplet[3];
        droplets[0] = new Droplet();
        droplets[1] = new Droplet();
        droplets[2] = new Droplet();

        gradTexture = new Texture2D(2048, 1, TextureFormat.Alpha8, false);
        gradTexture.wrapMode = TextureWrapMode.Clamp;
        gradTexture.filterMode = FilterMode.Bilinear;
        for (var i = 0; i < gradTexture.width; i++)
        {
            var x = 1.0f / gradTexture.width * i;
            var a = waveform.Evaluate(x);
            gradTexture.SetPixel(i, 0, new Color(a, a, a, a));
        }
```

Slika 79: Skripta RippleEffect.cs, varijable, UpdateShaderParameters funckija i Awake funkcija

```
        gradTexture.Apply();

        material = new Material(shader);
        material.hideFlags = HideFlags.DontSave;
        material.SetTexture("_GradTex", gradTexture);
        UpdateShaderParameters();
    }

    void Update() {
        if (dropInterval > 0)
        {
            timer += Time.deltaTime;
            while (timer > dropInterval) {
                timer -= dropInterval;
            }
        }
        foreach (var d in droplets) d.Update();
        UpdateShaderParameters();
    }

    void OnRenderImage(RenderTexture source, RenderTexture destination){
        Graphics.Blit(source, destination, material);
    }

    public void Emit(Vector2 pos){
        droplets[dropCount++ % droplets.Length].Reset(pos);
    }

    IEnumerator Stop(){
        yield return new WaitForSeconds(.3f);
    }
}
```

Slika 80: Skripta RippleEffect.cs, funckije Awake, Update, OnRenderImage, Emit i korutina Stop

7. SISTEM ČESTICA

Čestice (eng. particles) su male, jednostavne slike ili mreže koje se prikazuju i kreću u velikom broju pomoću sustava čestica. Svaka čestica predstavlja mali dio tekućeg ili amorfnog entiteta, a učinak svih čestica zajedno stvara dojam cjelokupnog entiteta. Primjerice, uz pomoć oblaka dima, svaka čestica imala bi malu dimnu teksturu sličnu sićušnom oblaku. Kada su mnogi od tih mini oblaka raspoređeni zajedno u nekom području scene, ukupni učinak je veći oblak punijeg volumena.

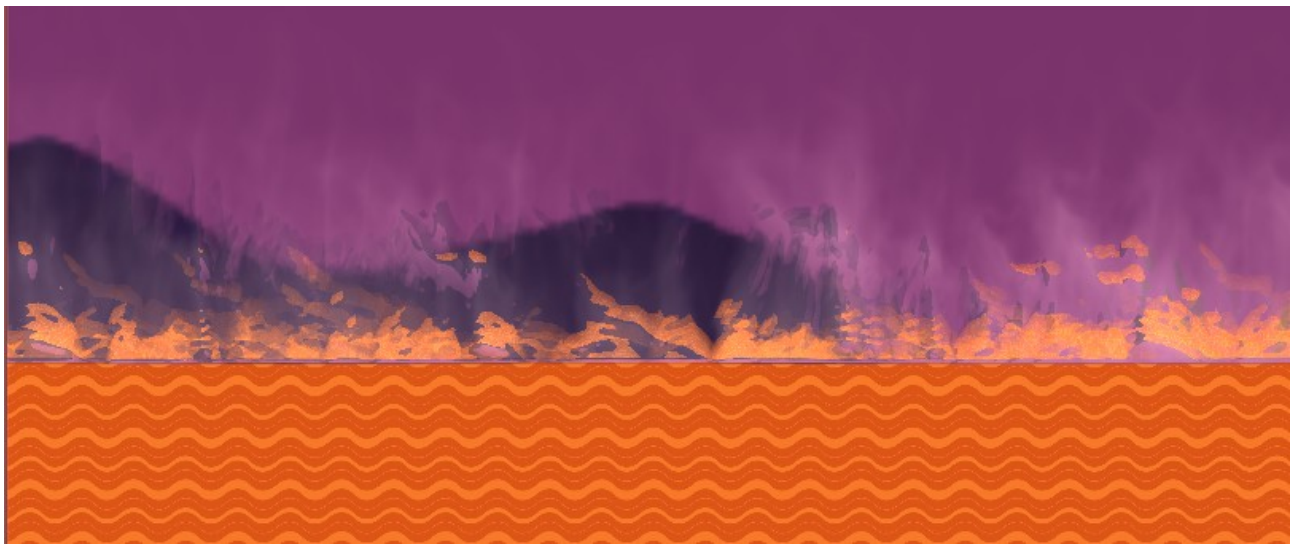
Svaka čestica ima unaprijed određeni vijek trajanja, obično nekoliko sekundi, tijekom kojih može proći različite promjene. Ona počinje svoj život kada se generira ili emitira iz sustava čestica. Sustav emitira čestice na slučajnim mjestima unutar područja prostora u obliku kugle, hemisfere, konusa, kutije ili bilo koje proizvoljne mreže (eng. mesh).

Čestica se prikazuje dok vrijeme ne istekne, u kojem trenutku je uklonjena iz sustava. Stopa emisije sustava otprilike pokazuje koliko se čestica emitira u sekundi, iako su točna vremena emisije slučajno odabrana. Izbor brzine emisije i prosječnog životnog vijeka čestica određuju broj čestica u "stabilnom" stanju (odnosno, gdje se emisija i smrt čestica događaju istom brzinom) i koliko dugo sustav treba da dođe do tog stanja.

Postavke emisije i životnog vijeka utječu na cjelokupno ponašanje sustava, ali se i pojedine čestice mogu mijenjati tijekom vremena. Svaka od njih ima vektor brzine koji određuje smjer i udaljenost čestica koja se pomiče sa svakim ažuriranjem slike u sekundi (eng. frame per second).

Brzina se može mijenjati silama i gravitacijom koju primjenjuje sam sustav ili kada čestice se čestice miču oko zone vjetra na terenu.
Boja, veličina i rotacija svake čestice mogu se mijenjati tijekom svog životnog vijeka ili proporcionalno njegovoj trenutnoj brzini kretanja. Boja uključuje alfa (transparentnost) komponentu, tako da se čestica može postupno uvenuti i nestati iz postojanja, umjesto da se jednostavno pojavljuje i nestaje naglo.

Korištena u kombinaciji, dinamika čestica može se koristiti za simulaciju mnogih vrsta fluidnih učinaka vrlo uvjerljivo. Na primjer, vodopad se može simulirati korištenjem tankog oblika ispuštanja i puštanja čestica vode jednostavno pada pod gravitaciju, ubrzavajući kako idu. Dim od požara ima tendenciju rasta, širenja i na kraju raspršivanja, tako da sustav treba koristiti silu prema gore na česticama dima i povećati njihovu veličinu i transparentnost tijekom njihovih života.



Slika 81: Primjer sustava čestica iz projekta, izvor: projekt

7.1 AUDIO

Audio korišten u projektu preuzet je iz besplatnih paketa Metal Mayhem Music Pack i Universal Sound FX iz Unity trgovine.

8. ZAKLJUČAK

Proces izrade video igara sastoji se od mnogo različitih dijelova kao što su; proces izrade likova i okoliša, animiranje likova, postavljanje scena u Unity razvojnome okruženju, animiranje objekata u okolišu, kodiranje mehanika igre, uljepšavanje izgleda igre pomoću alata kao što su efekti naknadne obrade na kameri ili pomoću sistema čestica.

Unity Engine je vrlo pristupačan alat za izradu video igara, za početne i napredne korisnike. Unity je među-platformsko razvojno okruženje za razvoj igara koje je razvijeno od strane Unity Technologies. Verzija programa koja se koristi za izradu ovoga projekta i rada jest Unity 2019.1. Programski jezik u projektu jest C#.

Mehanike igre su različiti sistemi koji ovise jedni o drugima kako bi pravilno funkcioniralo. Na primjer, sustav kretanja ne može pravilno funkcionirati bez sustava detekcije kolizije, dok je sustav kolizije sam po sebi beskoristan.

Animacije su izrađene od niza 2D slika, te se iste koriste u animator komponenti i tvore konačne automate stanja koja se pozivaju kroz skripte kada su određenu uvjeti zadovoljeni.

Igra je vrlo izazovna. Svaka razina se sastoji od niza zamki kroz koje igrač mora proći kako bi došao do portala koji ga vodi na sljedeću razinu. Igrač ima samo jedan život, te ako umre vraća se na početak razine na kojoj se trenutno nalazi.

9. LITERATURA

- Axon, Samuel (Rujan 27, 2016). *"Unity at 10: For better—or worse—game development has never been easier"*. [Online] Dostupno na: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/ [Pristupljeno: 06.07.2019]
- "What's new in Unity 5.0". Unity Technologies. Dostupno na: https://unity3d.com/unity/whats-new/unity-5.0 [Pristupljeno: 07.07.2019.]
- "*Celeste*", Wikipedia. Dostupno na: https://en.wikipedia.org/wikiCeleste_(video_game) , [Pristupljeno: 24.06.2019.]
- Matulef, Jeffrey (Srpanj 20, 2016). *"Towerfall dev's next game Celeste recalls Super Meat Boy"* [Online] Dostupno na: https://www.eurogamer.net/articles/2016-07-20-towerfall-devs-next-game-celeste-recalls-super-meat-boy , [Pristupljeno 26.06. 2019]
- "Ori and the Blind Forest Combines Beauty and Skill". *news.xbox.com*. Dostupno na: https://news.xbox.com/en-us/2014/06/11/games-ori-and-the-blind-forest-e3/ [Pristupljeno: 27.06.2019] .
- Leah B. Jackson (Lipanj 12, 2014). *"E3 2014: The Enchanting Beauty of Ori and The Blind Forest"*. [Online] *Dostupno na:* https://www.ign.com/articles/2014/06/12/e3-2014-the-enchanting-beauty-of-ori-and-the-blind-forest, *[Pristupljeno 28.06.2019]*
- Team Cherry 19. Studeni 2014)."HollowKnight" [Online], Dostupno na: https://www.kickstarter.com/projects/11662585/hollow-knight/description [Pristupljeno: 01.07.2019.]
- "*Hollow Knight*",Wikipedia. Dostupno na: https://en.wikipedia.org/wiki/Hollow_Knight , [Pristupljeno: 01.07.2019.])
- Whitaker, Jed (27. Ožujak 2017.). *"Review: Hollow Knight"* [Online], Dostupno na: https://www.destructoid.com/review-hollow-knight-427608.phtml [Pristupljeno: 01.07.2019]
- Unity documentation, verzija 2019.1, dostupno na: https://docs.unity3d.com/Manual/SpriteEditor.html , pristupljeno 17.06.2019

- Unity documentation, verzija 2019.1, dostupno na https://docs.unity3d.com/Manual/AnimationClips.html , pristupljeno 17.06.2019
- Unity documentation, dostupno na https://docs.unity3d.com/Manual/class-AnimatorController.html , pristupljeno: 23.06.2019
- Unity documentation, dostupno na https://docs.unity3d.com/Manual/StateMachineBasics.html , pristupljeno: 23.06.201
- Unity documentation, dostupno na: https://docs.unity3d.com/Manual/class-State.html , pristupljeno: 25.06.2019
- The Book of Shaders, Dostupno na: https://thebookofshaders.com/01/ [Pristupljeno: 14.07.2019]

Popis slika:

SAŽETAK

Cilj ovoga rada jest prikazati, opisati i objasnit proces izrade 2D platformer igre u Unity razvojnome okruženju. Korištenja verzija softvera jest Unity 2019.1. U radu je prikazan proces izrade i korištenja animacija, od animiranja likova do različitih objekata u okolišu, opisana je radnja igra, te prikazan programski kod iza mehanika unutar igre. Mehanike koje su prikazane su; kretanje, koje objedinjuje akcije kao što su; skakanje sa zida na zid, penjanje, odnosno klizanje po zidovima, nagli poleti na komandu igrača. Druge mehanike su; detekcija kolizije, dijalozi, zamke.

Ključne riječi: Unity, 2D, platformer, mehanike igre, programiranje

ABSTRACT

The aim of this paper is to present, describe and explain the process of creating a 2D platformer game in a Unity development environment. The software version used is Unity 2019.1. The paper describes the process of creating and using animations, from animating characters to various objects in the environment, describing the setting of the game, and the programming code behind the mechanics within the game. The mechanics shown are; movement, which integrates actions such as; jumping from wall to wall, climbing or sliding on walls, dashing at the command of a player. Other mechanics are; collision detection, dialogues, pitfalls.

Keywords: Unity, 2D, platformer, game mechanics, programming